1 Common Interface Functionality 3
2 Common Interface Functionality through Pexpect 9
3 Sage wrapper around pexpect's spawn class and 13
4 Interface to Axiom 15
5 The Elliptic Curve Factorization Method 21
6 Interface to 4ti2 27
7 Interface to FriCAS 33
8 Interface to Frobby for fast computations on monomial ideals. 45
9 Interface to GAP 49
  9.1 First Examples 49
  9.2 GAP and Singular 49
  9.3 Saving and loading objects 51
  9.4 Long Input 51
  9.5 Changing which GAP is used 51
10 Interface to GAP3 59
  10.1 Obtaining GAP3 59
  10.2 Changing which GAP3 is used 59
  10.3 Functionality and Examples 60
  10.4 Common Pitfalls 61
  10.5 Examples 62
11 Interface to Groebner Fan 67
12 Pexpect Interface to Giac 69
  12.1 Tutorial 70
13 Interface to the Gnuplot interpreter 79
14 Interface to the GP calculator of PARI/GP 81
15 Interface for extracting data and generating images from Jmol readable files. 91
30 Library interface to Maxima 247
31 Interface to MuPAD 257
32 Interface to mwrank 261
33 Interface to GNU Octave 265
   33.1 Computation of Special Functions 265
   33.2 Tutorial 266
34 Interface to PHC 271
35 Interface to polymake 279
37 Parallel Interface to the Sage interpreter 297
38 Interface to QEPCAD 299
39 Interface to Bill Hart’s Quadratic Sieve 325
40 Interfaces to R 329
41 Interface to several Rubik’s cube solvers. 343
42 Interface to Sage 347
43 Interface to Scilab 353
44 Interface to Singular 359
   44.1 Introduction 359
   44.2 Tutorial 360
   44.3 Computing the Genus 362
   44.4 An Important Concept 363
   44.5 Long Input 364
45 SymPy -> Sage conversion 381
46 The Tachyon Ray Tracer 383
47 Interface to TIDES 385
48 Interface to the Sage cleaner 389
49 Quitting interfaces 391
50 An interface to read data files 393
51 Indices and Tables 395
Python Module Index 397
Index 399
Sage provides a unified interface to the best computational software. This is accomplished using both C-libraries (see C/C++ Library Interfaces) and interpreter interfaces, which are implemented using pseudo-tty’s, system files, etc. This chapter is about these interpreter interfaces.

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

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

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

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

1. gap.console(): You are completely using another program, e.g., gap/magma/gp Here Sage is serving as nothing more than a convenient program launcher, similar to bash.

2. gap.interact(): This is a convenient way to interact with a running gap instance that may be “full of” Sage objects. You can import Sage objects into this gap (even from the interactive interface), etc.

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

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

AUTHORS:

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

```python
class sage.interfaces.interface.AsciiArtString
    Bases: str

class sage.interfaces.interface.Interface(name)
    Bases: sage.misc.fast_methods.WithEqualityById, sage.structure.parent_base.ParentWithBase
    Interface interface object.

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

```python
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.

```python
evaluate(*args, **kwds)
```

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

**EXAMPLES:**

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

```python
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.

```python
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:**

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

```python
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.

```python
help(s)
```

```python
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.

```python
name(new_name=None)
```

```python
new(code)
```

```python
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:**

```python
```
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

\textbf{read}(\textit{filename})

\textbf{EXAMPLES:}

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

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

\textbf{set}(\textit{var, value})

Set the variable \textit{var} to the given value.

\textbf{set_seed}(\textit{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.

\textbf{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):
  ... NotImplementedException: This interpreter did not implement a set_seed function

\textbf{class} sage.interfaces.interface.InterfaceElement\textbf{(parent, value, is_name=False, name=None)}

\textbf{Bases:} sage.structure.element.Element

Interface element.

\textbf{attribute}(\textit{attrname})

If this wraps the object \textit{x} in the system, this returns the object \textit{x}.\textit{attrname}. This is useful for some systems that have object oriented attribute access notation.
EXAMPLES:

```python
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]]
```

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

`bool()`
Convert this element to a boolean.

EXAMPLES:

```python
sage: singular(0).bool()
False
sage: singular(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:

```python
sage: a = maxima(str(2^1000))
sage: a.get_using_file()
˓→'10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251...
˓→'
```

`hasattr(attrname)`
Returns whether the given attribute is already defined by this object, and in particular is not dynamically generated.

EXAMPLES:

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

`is_string()`
Tell whether this element is a string.

By default, the answer is negative.

`name(new_name=None)`
Returns 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: x = r([1,2,3]); x  # optional - rpy2
[1] 1 2 3
sage: x.name()         # optional - rpy2
'sage...
```

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

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

```
sage(*args, **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]
```

```
class sage.interfaces.interface.InterfaceFunction
```

Bases: sage.structure.sage_object.SageObject

Interface function.

```
class sage.interfaces.interface.InterfaceFunctionElement
```

Bases: sage.structure.sage_object.SageObject

Interface function element.

```
help()
```

```
sage.interfaces.interface.is_InterfaceElement(x)
```
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.
- Simon King (2010-11-23): Ensure that the interface is started again after a crash, when a command is executed in _eval_line. Allow synchronisation of the GAP interface.
- François Bissey, Bill Page, Jeroen Demeyer (2015-12-09): Upgrade to pexpect 4.0.1 + patches, see trac ticket #10295.

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

Bases: `sage.interfaces.interface.Interface`

Expect interface object.

**clear_prompts()**

- Returns the command used in this interface.

**EXAMPLES:**

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

**detach()**

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

**EXAMPLES:**

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

Calling detach() a second time does nothing:

sage: maxima.detach()

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

INPUT:

• code – text to evaluate

• strip – bool; whether to strip output prompts, etc. (ignored in the base class).

• locals – None (ignored); this is used for compatibility with the Sage notebook’s generic system interface.

• allow_use_file – bool (default: True); if True and code exceeds an interface-specific threshold then code will be communicated via a temporary file rather 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:
quit(\texttt{verbose=False})

Quit the running subprocess.

INPUT:

\begin{itemize}
  \item verbose – (boolean, default \texttt{False}) print a message when quitting this process?
\end{itemize}

EXAMPLES:

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

Calling \texttt{quit()} a second time does nothing:

\begin{verbatim}
sage: maxima.quit(\texttt{verbose=True})
\end{verbatim}

server()

Returns the server used in this interface.

EXAMPLES:

\begin{verbatim}
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'
\end{verbatim}

set_server_and_command(\texttt{server=None}, \texttt{command=None}, \texttt{server_tmpdir=None}, \texttt{ulimit=None})

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

EXAMPLES:

\begin{verbatim}
sage: magma.set_server_and_command(server = 'remote', command = 'mymagma')  # indirect doctest
No remote temporary directory (option server_tmpdir) specified, using /tmp/ on remote
sage: magma.server()
'remote'
sage: magma.command()
"sage-native-execute ssh -t remote 'mymagma'"
\end{verbatim}
user_dir()

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

class sage.interfaces.expect.ExpectFunction(parent, name)
    Bases: sage.interfaces.interface.InterfaceFunction
    Expect function.

class sage.interfaces.expect.FunctionElement(obj, name)
    Bases: sage.interfaces.interface.InterfaceFunctionElement
    Expect function element.

class sage.interfaces.expect.StdOutContext(interface, silent=False, stdout=None)
    Bases: object
    A context in which all communication between Sage and a subprocess interfaced via pexpect is printed to stdout.

code: sage.interfaces.expect.console(cmd)

class sage.interfaces.expect.gc_disabled
    Bases: object
    This is a “with” statement context manager. Garbage collection is disabled within its scope. Nested usage is properly handled.

EXAMPLES:

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

sage.interfaces.expect.is_ExpectElement(x)
SAGE WRAPPER AROUND PEXPECT’S Spawn CLASS AND

the ptyprocess’s PtyProcess class.

AUTHOR:

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

```python
class sage.interfaces.sagespawn.SagePtyProcess(pid, fd)
Bases: ptyprocess.ptyprocess.PtyProcess

close(force=None)
    Quit the child process: send the quit string, close the pseudo-tty and kill the process.
    This function returns immediately, it doesn’t wait for the child process to die.

EXAMPLES:

sage: from sage.interfaces.sagespawn import SageSpawn
sage: s = SageSpawn("sleep 1000")

sage: s.close()

sage: while s.isalive():  # long time (5 seconds)
    ....:     sleep(float(0.1))
```

terminate_async(interval=5.0)

Terminate the child process group asynchronously.

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

INPUT:

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

EXAMPLES:

Run an infinite loop in the shell:

```python
sage: from sage.interfaces.sagespawn import SageSpawn

sage: s = SageSpawn("sh", ['"-c", "while true; do sleep 1; done"'])

Check that the process eventually dies after calling terminate_async:

```

(continues on next page)
....:       sleep(float(0.1))
....:   else:
....:       break  # process got killed

class sage.interfaces.sagespawn.SageSpawn(*args, **kwds)
    Bases: pexpect.pty_spawn.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 ...

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"])
    sage: _ = E.expect_peek("w")
    sage: 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'
CHAPTER
FOUR

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.


• 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.

<table>
<thead>
<tr>
<th>sage: axiom('3 * 5')</th>
<th>#optional - axiom</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sage: a = axiom(3) * axiom(5); a</th>
<th>#optional - axiom</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

The type of a is AxiomElement, i.e., an element of the axiom interpreter.
The underlying Axiom type of \(a\) is also available, via the type method:

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

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

```python
sage: F = axiom.factor('x^5 - y^5'); F  #optional - axiom
- (y - x)(y + x y + x y + x y + x )
```

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

```python
sage: a = axiom(2/3); a  #optional - axiom
2
-3
```

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.

```python
sage: print(axiom.eval('factor(x^5 - y^5)'))  # optional - axiom
- (y - x)(y + x y + x y + x y + x )
```

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.

```python
sage: f = axiom('x^5 - y^5')  #optional - axiom
sage: f^2  #optional - axiom
y - 2x 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

sage: axiom('1/100 + 1/101')  #optional - axiom
201
----
10100
sage: 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,
service_tmpdir=None, init_code=['\lisp (si::readline-off)'])

Bases: sage.interfaces.axiom.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.
-----------------------------------------------

sage.interfaces.axiom.AxiomElement
alias of sage.interfaces.axiom.PanAxiomElement

sage.interfaces.axiom.AxiomExpectFunction
alias of sage.interfaces.axiom.PanAxiomExpectFunction

sage.interfaces.axiom.AxiomFunctionElement
alias of sage.interfaces.axiom.PanAxiomFunctionElement

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

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

Interface to a PanAxiom interpreter.

get(var)
Get the string value of the Axiom variable var.

EXAMPLES:

sage: axiom.set('xx', '2')  #optional - axiom
sage: axiom.get('xx')  #optional - axiom
'2'
set(var, value)
Set the variable var to the given value.

EXAMPLES:

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

class sage.interfaces.axiom.PanAxiomElement
Bases: sage.interfaces.expect.ExpectElement

as_type(type)
Returns self as type.

EXAMPLES:

```python
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
```

coma(*args)
Returns a Axiom tuple from self and args.

EXAMPLES:

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

type()
Returns the type of an AxiomElement.

EXAMPLES:

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

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

EXAMPLES:

```python
sage: a = axiom(x^2+1); a         #optional - axiom
2
```
class sage.interfaces.axiom.PanAxiomExpectFunction(parent, name)
    Bases: sage.interfaces.expect.ExpectFunction

class sage.interfaces.axiom.PanAxiomFunctionElement(object, name)
    Bases: sage.interfaces.expect.FunctionElement

sage.interfaces.axiom.axiom_console()
    Spawn a new Axiom command-line session.

    EXAMPLES:

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

sage.interfaces.axiom.is_AxiomElement(x)
    Returns True if x is of type AxiomElement.

    EXAMPLES:

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

sage.interfaces.axiom.reduce_load_Axiom()
    Returns the Axiom interface object defined in sage.interfaces.axiom.

    EXAMPLES:

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

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 \texttt{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 http://ecm.gforge.inria.fr 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.

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

\texttt{Bases: sage.structure.sage_object.SageObject}

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

See http://ecm.gforge.inria.fr

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 found number 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

```python
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** – a 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:**
A 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
```

**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** – a 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, 636680576909027985741435139224233]
```

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
```

**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]
sage: 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
```

**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:**

- $n$ – a 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 \(n = p \cdot q\). 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
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: f = ECM()
sage: n = 4321328878839031080009802143314445113500016816977037257
sage: f.one_curve(n, B1=500000, algorithm="P-1")
[67872792749691946529, 6366805760909027985741435139224233]
sage: f = ECM()
sage: n = 2088352670731726254864791941658631875815083
sage: f.one_curve(n, B1=2000, algorithm="P+1", x0=5)
[328006342451, 6366805760909027985741435139224233]
```

**recommended_B1(factor_digits)**
Return recommended \(B1\) setting.

**INPUT:**
- `factor_digits` – integer. Number of digits.

**OUTPUT:**

**EXAMPLES:**

```sage
sage: ecm.recommended_B1(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` – a 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 30448163954141809957445496544854621998616257489887231115912293
→ (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
Computation 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
Computation roots of G took 8ms
Building G from its roots took 16ms
Computation roots of G took 7ms
Building G from its roots took 16ms
Computation G * H took 6ms
Reducing G * H mod F took 5ms
Computation roots of G took 7ms
Building G from its roots took 17ms
Computation G * H took 5ms
Reducing G * H mod F took 5ms
Computation polyeval(F,G) took 34ms
Computation 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 114.21d 7.27y 196.42y 337811y 5e+10y 7e+15y Inf
Expected curves: 4911, Expected time: 32.25m
http://www.4ti2.de

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:**

- Bjarke Hammersholt Roune (2009-06-26): Added Groebner, made code usable as part of the Sage library and added documentation and some doctests.

```python
class sage.interfaces.four_ti_2.FourTi2(directory=None)
  Bases: object

This object defines an interface to the program 4ti2. Each command 4ti2 has is exposed as one method.

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 – A 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]
```

```python
circuits(mat=None, project=None)
  Run the 4ti2 program circuits on the parameters. See http://www.4ti2.de/ 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]]

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/"
)'

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

Run the 4ti2 program graver on the parameters. See http://www.4ti2.de/ 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],
 [ 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]]

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 http://www.4ti2.de/ 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]]

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

Run the 4ti2 program hilbert on the parameters. See http://www.4ti2.de/ for details.

EXAMPLES:
```python
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]
```

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

Run the 4ti2 program minimize on the parameters. See [http://www.4ti2.de/](http://www.4ti2.de/) for details.

**EXAMPLES:**

```python
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.
```

**ppi** *(n)*

Run the 4ti2 program ppi on the parameters. See [http://www.4ti2.de/](http://www.4ti2.de/) for details.

**EXAMPLES:**

```python
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]]
```

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

Run the 4ti2 program qsolve on the parameters. See [http://www.4ti2.de/](http://www.4ti2.de/) for details.

**EXAMPLES:**

```python
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]]
```

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

Run the 4ti2 program rays on the parameters. See [http://www.4ti2.de/](http://www.4ti2.de/) for details.

**EXAMPLES:**

```python
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 1 0 0 2 1]]
```
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 $\mathbb{Z}$.

EXAMPLES:

```python
sage: from sage.interfaces.four_ti_2 import four_ti_2
sage: four_ti_2.read_matrix("test_file")
```

write_array(array, nrows, ncols, filename)
Write the matrix array of integers (can be represented as a list of lists) to the file filename in directory() in 4ti2 format. The matrix must have nrows rows and ncols columns.

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:

```python
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")
```

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:
write_single_row(row, filename)
Write the list row to the file filename in 4ti2 format as a matrix with one row.

INPUT:
- row – A list of integers.
- filename – A file name not including a path.

EXAMPLES:

zsolve(mat=None, rel=None, rhs=None, sign=None, lat=None, project=None)
Run the 4ti2 program zsolve on the parameters. See http://www.4ti2.de/ for details.

EXAMPLES:

sage: from sage.interfaces.four_ti_2 import four_ti_2
sage: four_ti_2.zsolve(lat=[[1,2,3],[1,1,1]]) # optional - 4ti2
[ [1 2 3]
  [0 0 0], [], [1 1 1] ]
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
sage: fricas('3 * 5')  # optional -
˓→fricas
15
sage: a = fricas(3) * fricas(5); a  # optional -
˓→fricas
15
```

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

```sage
sage: type(a)  # optional -
˓→fricas
<class 'sage.interfaces.fricas.FriCASElement'>
sage: a.parent()  # optional -
˓→fricas
FriCAS
```

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

```sage
sage: a.typeOf()  # optional -
˓→fricas
PositiveInteger
```

FriCAS objects are normally displayed using “ASCII art”:
Functions defined in FriCAS are available as methods of the `fricas` object:

```python
sage: F = fricas.factor('x^5 - y^5'); F
fricas
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))
```

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:

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

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

```python
sage: 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:

```python
sage: f = fricas('(x^5 - y^5)^10000')
fricas
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:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)" # optional -
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```

FriCAS can solve linear ordinary differential equations:

```plaintext
guessADE([partition n for n in 0..40], homogeneous==4)
```
FriCAS can expand expressions into series:

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

sage: a.coefficients()[38].sage()  # optional ~
   # fricas
   -2947202633533727150423659490832640468979/
   27421448206632936368243066750897974998466560000000

sage: ex = sqrt(atan(x)); a = fricas(ex).series(x=0); a  # optional ~
   1 2 3 4
   1 - - - x - - x + O(x )
   6 360

sage: a.coefficient(9/2).sage()  # optional ~
   # fricas
   31/360

sage: x = fricas("x::TaylorSeries Fraction Integer")  # optional ~
   # fricas
sage: y = fricas("y::TaylorSeries Fraction Integer")  # optional ~
   # fricas
sage: 2*(1+2*x+sqrt(1-4*x)-2*x*y).recip()  # optional ~
   # fricas
   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)
```

FriCAS does some limits right:

```
sage: x = var('x'); ex = x^4*2*exp(-x)*Ei(x) - x; fricas(ex).limit(x=oo)  # optional ~
   # fricas
   1
```
class sage.interfaces.fricas.FriCAS(name='fricas', command='fricas -nosman',
        script_subdirectory=None, logfile=None, server=None,
        server_tmpdir=None)

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

Interface to a FriCAS interpreter.

console()
Spawn a new FriCAS command-line session.

EXAMPLES:

```sage
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.
----------------------------------------------
```

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 – bool; remove the output markers when True.

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

EXAMPLES:

```sage
sage: fricas.set("x", "1783"); fricas("x")  # optional - → fricas
1783
sage: fricas("x")  # optional - → fricas
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
sage: fricas.set('xx', '2')  # optional - → fricas
sage: fricas('xx')  # optional - → fricas
'2'
```
sage: a = fricas('(1 + sqrt(2))^5')  # optional - fricas
sage: fricas.get(a.name())          # optional - fricas

sage: fricas.get('(1 + sqrt(2))^5') # optional - fricas

sage: fricas.new('(1 + sqrt(2))^5') # optional - fricas

sage: fricas.get('(1 + sqrt(2))^5') # optional - fricas

sage: fricas.get((1 + sqrt(2))^5)  # optional - fricas

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')         # optional - fricas
sage: fricas.get('xx')              # optional - fricas

class sage.interfaces.fricas.FriCASElement(parent, value, is_name=False, name=None)
Bases: sage.interfaces.expect.ExpectElement
Instances of this class represent objects in FriCAS.

Using the method \texttt{sage()} we can translate some of them to SageMath objects:

\_\texttt{sage\_}()

Convert \texttt{self} to a Sage object.

**EXAMPLES:**

**Floats:**

\begin{verbatim}
sage: fricas(2.1234).sage()          # optional - fricas
   \rightarrow fricas
2.12340000000000
sage: _.parent()                   # optional - fricas
   \rightarrow fricas
Real Field with 53 bits of precision
sage: a = RealField(100)(pi)        # optional - fricas
sage: fricas(a).sage()              # optional - fricas
3.1415926535897932384626433833
sage: _.parent()                   # optional - fricas
Real Field with 100 bits of precision
sage: fricas(a).sage() == a         # optional - fricas
True
sage: fricas(2.0).sage()            # optional - fricas
2.00000000000000
sage: _.parent()                   # optional - fricas
Real Field with 53 bits of precision
\end{verbatim}

**Algebraic numbers:**

\begin{verbatim}
sage: a = fricas('(1 + sqrt(2))^5'); a
   # optional - fricas
+-+
29 \|2 + 41
sage: b = a.sage(); b              # optional - fricas
82.0121933088198?
sage: b.radical_expression()      # optional - fricas
29*sqrt(2) + 41
\end{verbatim}

**Integers modulo \(n\):**

\begin{verbatim}
sage: fricas("((42^17)^1783)::IntegerMod(5^(5^5))").sage() == Integers(5^(5^ 5))((42^17)^1783) # optional - fricas
True
\end{verbatim}

**Matrices over a prime field:**
We can also convert FriCAS’s polynomials to Sage polynomials:

```
sage: a = fricas("x^2 + 1"); a.typeOf()  # optional -
    → fricas
Polynomial(Integer)
sage: a.sage()  # optional -
    → fricas
x^2 + 1
sage: _.parent()  # optional -
    → fricas
Univariate Polynomial Ring in x over Integer Ring
```

```
sage: fricas("x^2 + y^2 + 1/2").sage()  # optional -
    → fricas
y^2 + x^2 + 1/2
sage: _.parent()  # optional -
    → fricas
Multivariate Polynomial Ring in y, x over Rational Field
```

```
sage: fricas("1\$Polynomial Integer").sage()  # optional -
    → fricas
1
```

```
sage: fricas("x^2/2").sage()  # optional -
    → fricas
1/2*x^2
```

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

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

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

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

Rational functions:
Expressions:

```sage
sage: fricas('x^2 + 1/z').sage()   # optional - fricas
x^2 + 1/z
```

```sage
sage: fricas(pi).sage()   # optional - fricas
pi
```

```sage
sage: fricas('sin(x+y)/exp(z)*log(1+e^z)').sage()   # optional - fricas
e^(-z)*log(e + 1)*sin(x + y)
```

```sage
sage: fricas('factorial(n)').sage()   # optional - fricas
factorial(n)
```

```sage
sage: fricas('integrate(sin(x+y), x=0..1)').sage()   # optional - fricas
-cos(y + 1) + cos(y)
```

```sage
sage: fricas('integrate(x*sin(1/x), x=0..1)').sage()   # optional - fricas
'failed'
```

```sage
sage: fricas('integrate(sin((x^2+1)/x),x)').sage()   # optional - fricas
integral(sin((x^2 + 1)/x), x)
```

Todo:
- Converting matrices and lists takes much too long.

Matrices:

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

```sage
sage: fricas("[matrix [[i for i in 1..n]] for n in 0..5]" s).sage()   # optional - fricas, long time
[[], [1], [1 2], [1 2 3], [1 2 3 4], [1 2 3 4 5]]
```
Error handling:

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

```sage
sage: s = fricas("series(sqrt(1+x), x=0)"); s  # optional -
→ fricas
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
sage: s.sage()  # optional -
→ fricas
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 .
```

**bool()**

Coerce the expression into a boolean.

**EXAMPLES:**

```sage
sage: fricas("1=1").bool()  # optional -
→ fricas
True
```

```sage
sage: fricas("1~=1").bool()  # optional -
→ fricas
```

(continues on next page)
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: sage.interfaces.expect.ExpectFunction
    Translate the pythonized function identifier back to a FriCAS operation name.

class sage.interfaces.fricas.FriCASFunctionElement(object, name)
    Bases: sage.interfaces.expect.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.

sage.interfaces.fricas.is_FriCASElement(x)
    Return True if x is of type FriCASElement.

    EXAMPLES:

    sage: from sage.interfaces.fricas import is_FriCASElement  # optional - → fricas
    sage: is_FriCASElement(fricas(2))                          # optional - → fricas
    True
    sage: is_FriCASElement(2)                                 # optional - → fricas
    False

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  # optional - → fricas
    sage: reduce_load_fricas()                                  # optional - → fricas
    FriCAS
CHAPTER EIGHT

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: (a, b, c, d) = QQ['a,b,c,d'].gens()  # optional - frobby
    sage: id = ideal(a * b, b * c, c * d, d * a)  # optional - frobby
    sage: alexander_dual = frobby.alexander_dual(id)  # optional - frobby
    sage: true_alexander_dual = ideal(b * d, a * c)  # optional - frobby
    sage: alexander_dual == true_alexander_dual  # use sets to ignore order #.  # optional - frobby
    True
We see how it is much faster to compute this with frobby than the built-in procedure for simplicial complexes.

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

```python
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]
```

**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:

```python
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
```

**hilbert** *(monomial_ideal)*

Computes 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 \(x_1, x_2, ..., x_n\). 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:

```python
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^10*b^2*c + b^10 + 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.

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

```
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 trac ticket #3028).

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

```
True
```

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

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

```
True
```
CHAPTER
NINE

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 \( f = \text{gap.new}(10) \), then \( f.\text{Factors()} \) returns the prime factorization of 10 computed using GAP.

9.1 First Examples

We factor an integer using GAP:

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

9.2 GAP and Singular

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

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

(continues on next page)
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.

```python
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
```

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

```python
sage: R = gap.PolynomialRing('Rationals', 2); R
PolynomialRing( Rationals, ["x_1", "x_2"] )
sage: 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 \(I.name()\).

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

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

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

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

```python
sage: f.Value(I, [1,2])
3
sage: g(1,2)  # agrees
3
```
9.3 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.

9.4 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)
```

9.5 Changing which GAP is used

Use this code to change which GAP interpreter is run. E.g.,

```
import sage.interfaces.gap
sage.interfaces.gap.gap_cmd = "/usr/local/bin/gap"
```

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:** sage.interfaces.gap.Gap_generic

Interface to the GAP interpreter.

**AUTHORS:**
- William Stein and David Joyner

```
compact()
```

Spawn a new GAP command-line session.

**EXAMPLES:**
**cputime**(t=None)

Returns 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:**

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

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

Get the string representation of the variable \( var \).

**EXAMPLES:**

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

```python
sage: gap = Gap(env={"LC_CTYPE": 'en_US.UTF-8'})
sage: print(gap.help('SymmetricGroup', pager=False))
50.1-... SymmetricGroup
→ SymmetricGroup( [filt, ]deg ) -------------- function ...
```

**save_workspace()**

Save the GAP workspace.

**set**(var, value)

Set the variable \( var \) to the given value.

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

```python
set_seed(seed=None)
```

Set the seed for gap interpreter.

The seed should be an integer.

**EXAMPLES:**

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

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

Bases: `sage.interfaces.gap.GapElement_generic`

**str**(use_file=False)

**EXAMPLES:**

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

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

Bases: `sage.structure.element.ModuleElement`, `sage.interfaces.tab_completion.ExtraTabCompletion`, `sage.interfaces.expect.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

```python
is_string()
```

Tell whether this element is a string.

**EXAMPLES:**

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

```python
class sage.interfaces.gap.GapFunction(parent, name)
```

Bases: `sage.interfaces.expect.ExpectFunction`

```python
class sage.interfaces.gap.GapFunctionElement(obj, name)
```

Bases: `sage.interfaces.expect.FunctionElement`

9.5. Changing which GAP is used
class sage.interfaces.gap.Gap_generic(name, prompt, command=None, server=None, server_tmpdir=None, ulimit=None, maxread=None, script_subdirectory=None, restart_on_ctrlc=False, verbose_start=False, init_code=[], max_startup_time=None, logfile=None, eval_using_file_cutoff=0, do_cleaner=True, remote_cleaner=False, path=None, terminal_echo=True):

Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.expect.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 - bool (default: True); if False, remove all backslash-newlines inserted by the GAP output formatter.
• strip - ignored
• split_lines – bool (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\nPrint(6);')
'46'
sage: gap.eval('Print("#"); Print(6);')
'#6'
sage: gap.eval('4; \n 6;')
'4\n6'
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\\n \\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')


function_call(function, args=None, kwds=None)
Calls the GAP function with args and kwds.

EXAMPLES:
get_record_element(record, name)
Return the element of a GAP record identified by name.

INPUT:
• record – a GAP record
• name – str

OUTPUT:
• GapElement

EXAMPLES:

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

```python
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]
```

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:
version()

Returns the version of GAP being used.

EXAMPLES:

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

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

AUTHOR:

- David Joyner and William Stein

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

INPUT:

- `x` – Gap integer mod ring element

EXAMPLES:

```python
sage: a = gap(Mod(3, 18)); a
ZmodnZObj( 3, 18 )
sage: b = sage.interfaces.gap.intmod_gap_to_sage(a); b
3
sage: b.parent()
Ring of integers modulo 18
sage: a = gap(Mod(3, 17)); a
Z(17)
sage: b = sage.interfaces.gap.intmod_gap_to_sage(a); b
3
sage: b.parent()
Finite Field of size 17
sage: a = gap(Mod(0, 17)); a
0*Z(17)
sage: b = sage.interfaces.gap.intmod_gap_to_sage(a); b
0
sage: b.parent()
Finite Field of size 17
sage: a = gap(Mod(3, 65537)); a
ZmodpZObj( 3, 65537 )
sage: b = sage.interfaces.gap.intmod_gap_to_sage(a); b
3
sage: b.parent()
Ring of integers modulo 65537
```
sage.interfaces.gap.is_GapElement(x)
Returns True if x is a GapElement.

EXAMPLES:

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

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

EXAMPLES:

```
sage: from sage.interfaces.gap import reduce_load_GAP
sage: reduce_load_GAP()
Gap
```
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 trac ticket #20107 and also

https://webusers.imj-prg.fr/~jean.michel/gap3/

10.1 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.
  

10.2 Changing which GAP3 is used

Warning: There is a bug in the pexpect module (see trac ticket #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
10.3 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: a = gap3('3+2')                      #optional - gap3
sage: a                                   #optional - gap3
5
sage: type(a)                             #optional - gap3
<class 'sage.interfaces.gap3.GAP3Element'>
```

```
sage: S5 = gap3('SymmetricGroup(5)')       #optional - gap3
sage: S5                                   #optional - gap3
Group( (1,5), (2,5), (3,5), (4,5) )
```

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: S5.Size()                           #optional - gap3
120
sage: S5.CharTable()                     #optional - gap3
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) ] ) ]
```

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

sage: S5.identity()                        #optional - gap3
()

sage: S5.degree                            #optional - gap3
5

sage: S5.1                                 #optional - gap3
(1,5)

sage: S5.2                                 #optional - gap3
(2,5)
```

2. By typing `%gap3` or `gap3.interact()` at the command-line, you can interact directly with the underlying GAP3 session.
Sage 9.5 Reference Manual: Interpreter Interfaces, Release 9.5

sage: gap3.interact()  #not tested

--> Switching to Gap3 <--

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

sage: gap3.console()  #not tested

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...

10.4 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"

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

Load a GAP3 package:

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

```
sage: gap3.load_package("chevie")             #optional - gap3
sage: gap3.version()                           # optional - gap3
'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: L = gap3([1,2,3])                          #optional - gap3
sage: L[1]                                       #optional - gap3
    1
    2
sage: 3 in L                                     #optional - gap3
    True
sage: 4 in L                                     #optional - gap3
    False
sage: m = gap3([[1,2],[3,4]])                    #optional - gap3
sage: m[2,1]                                     #optional - gap3
    3
sage: [1,2] in m                                 #optional - gap3
    True
sage: [3,2] in m                                 #optional - gap3
    False
sage: gap3([1,2]) in m                          #optional - gap3
    True
```

Controlling variable names used by GAP3:

```
sage: gap3('2', name='x')                       #optional - gap3
    2
sage: gap3('x')                                 #optional - gap3
    2
sage: gap3.unbind('x')                          #optional - gap3
sage: gap3('x')                                 #optional - gap3
    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: sage.interfaces.gap.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` – bool (default: False); if True, then `value` is the variable name for the object
• `name` – str (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:**

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

**AUTHORS:**

• Franco Saliola (Feb 2010)

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

Bases: `sage.interfaces.gap3.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:**

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

```python
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
```

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

Bases: sage.interfaces.gap.Gap_generic

A simple Expect interface to GAP3.

EXAMPLES:

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

AUTHORS:

• Franco Saliola (Feb 2010)

console()

Spawn a new GAP3 command-line session.

EXAMPLES:

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

(continues on next page)
cputime$(t=None)$

Returns 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:

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

help$(\text{topic}, \text{pager=}\text{True})$

Print help on the given topic.

INPUT:

- topic – string

EXAMPLES:

```python
sage: gap3.help('help', pager=False) #optional - gap3
Help _______________________________________________________
This section describes together with the following section... help system. The help system lets you read the manual inter...
```

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

\texttt{sage.interfaces.gap3.gap3\_console()}$

Spawn a new GAP3 command-line session.

EXAMPLES:

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

(continues on next page)
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 -w
--> gap3
'lib: v3r4p4 1997/04/18, src: v3r4p0 1994/07/10, sys: usg gcc ansi'
```
INTERFACE TO GROEBNER FAN

AUTHOR:

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

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

- at most 52 variables:
  - use gfan_substitute to make easier (?)
    MH: I think this is now irrelevant since gfan can accept the original ring variables
- --symmetry is really useful
  - permutations are 0-based *not* cycle notation; a <--> 0
    output is broken up much more nicely.
- -- can work in Z/pZ for p <= 32749
- -- can compute individual GB's for lex and revlex (via buchberger)

```
class sage.interfaces.gfan.Gfan
    Bases: object

    Interface to Anders Jensen's Groebner Fan program.
```
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](https://www-fourier.ujf-grenoble.fr/~parisse/giac.html)

Type `giac.[]` for a list of all the functions available from your Giac install. Type `giac.[]?` for Giac’s help about a given function. Type `giac(...)`, `giac.eval(...)`, `giac.ifactor(2005)` 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))))
```

If the string “error” (case insensitive) occurs in the output of anything from Giac, a RuntimeError exception is raised.
12.1 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.

12.1.1 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

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

We can write that in sage as

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

12.1.2 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```
```python
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```
```python
sage: giac( 'x^28-1' ).factor()
(x-1)*(x+1)*(x^2+1)*(x^4-x^2-2-x^1)*(x^2-x+1)*(x^4-x^2+1)(x^6-x^5+x^4-x^3-x^2-2-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)
```
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
```

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
```

```
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
```

12.1.4 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
```

More complicated programs should be put in a separate file and loaded.
12.1.5 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': 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_.

```python
class sage.interfaces.giac.Giac(maxread=None, script_subdirectory=None, server=None, server_tmpdir=None, logfile=None):
    Bases: sage.interfaces.expect.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])
sage: l.factors()
[y+sqrt(2),4]
sage: giac('assume(y>0)'); giac('y^2=3').solve('y')
y...
[sqrt(3)]
```
```
```
```
```
Polynomials or elements of SR can be evaluated directly by the giac interface.

Variable names in python and giac are independent:

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

```
completions(s)
Return all commands that complete the command starting with the string s.

EXAMPLES:
```
cputime$(t=None)$
Returns 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:

```python
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
```

eval$(\text{code}, \text{strip}=True, **\text{kwds})$
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:

```python
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
```

expect() 
Returns the pexpect object for this Giac session.

EXAMPLES:

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

get$(\text{var})$
Get the value of the variable var.

EXAMPLES:
help(string)
Display Giac help about string.
This is the same as typing "?string" in the Giac console.

INPUT:
• string – a 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))...
```

set(var, value)
Set the variable var to the given value.

EXAMPLES:

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

version()
Wrapper for giac’s version().

EXAMPLES:

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

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

Returns the definite integral if xmin is not None, otherwise returns 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
```


```python
sage: f = giac("exp(x^2)").integral('x',0,1) ; f
1.46265174...
```

```python
sage: x,y=giac('x'),giac('y');integrate(cos(x+y),'x=0..pi').simplify()
-2*sin(y)
```

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

```python
sage: f.diff('y').simplify()
sin(2*y)/y
```

```python
sage: f = giac("exp(x^2)").integral('x',0,1) ; f
1.46265174...
```

```python
sage: x,y=giac('x'),giac('y');integrate(cos(x+y),'x=0..pi').simplify()
-2*sin(y)
```

```python
sum(var=None, min=None, max=None)
Return the sum of self with respect to the variable x.
```

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

```python
unapply(var)
Creates a Giac function in the given arguments from a Giac symbol.
```

```python
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
```
sage.interfaces.giac.GiacFunction(parent, name)
Bases: sage.interfaces.expect.ExpectFunction

class sage.interfaces.giac.GiacFunctionElement(obj, name)
Bases: sage.interfaces.expect.FunctionElement

sage.interfaces.giac.giac_console()
Spawn a new Giac command-line session.

EXAMPLES:

```python
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

sage.interfaces.giac.reduce_load_Giac()
Returns the giac object created in sage.interfaces.giac.

EXAMPLES:

```python
sage: from sage.interfaces.giac import reduce_load_Giac
sage: reduce_load_Giac()
Giac
```
class sage.interfaces.gnuplot.Gnuplot
    Bases: sage.structure.sage_object.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 - (int) number of sample points to use
    • title - (string) title of the graph.
EXAMPLES:

```python
sage: 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
FOURTEEN

INTERFACE TO THE GP CALCULATOR OF PARI/GP

Type `gp` for a list of all the functions available from your Gp install. Type `gp` 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]
M
[1, 2; 3, 4]
M * M
[7, 10; 15, 22]
M + M
[2, 4; 6, 8]
M.matdet()
-2

E = gp.ellinit([1,2,3,4,5])
E.ellglobalred()
[10351, [1, -1, 0, -1], 1, [11, 1; 941, 1], [[1, 5, 0, 1], [1, 5, 0, 1]]
E.ellan(20)
[1, 1, -1, -3, 0, -1, -3, -3, -1, 0, 1, -1, 0, -1, 5, -3, 4, 3]

primitive_root(7)
3
x = gp("znlog( Mod(2,7), Mod(3,7))")
3^x % 7
2

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.

a = intnum(x=0,6,sin(x))
0.03982971334963397945434770208 # 32-bit
0.039829713349633979454347702077075594548 # 64-bit

a = gp("a")
0.03982971334963397945434770208 # 32-bit
0.039829713349633979454347702077075594548 # 64-bit
```

(continues on next page)
sage: gp.kill("a")
sage: gp("a")
a
Note that gp ASCII plots do work in Sage, as follows:

```python
sage: print(gp.eval("plot(x=0,6,sin(x))"))
```

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

```python
sage: t = '%s'*10**10000 # ten thousand character string.
sage: a = gp.eval(t)
sage: a = gp(t)
```

In Sage, the PARI large Galois groups datafiles should be installed by default:

```python
sage: f = gp('x^9 - x - 2')
sage: f.polygonal()
[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: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.expect.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` (int, 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` (int, default 1024) – length of initial list of local variables.
- `seed` (int, default random) – random number generator seed for pari

EXAMPLES:

```python
sage: Gp()
PARI/GP interpreter console()
Spawn a new GP command-line session.

EXAMPLES:

```python
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)
```python

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:

```python
sage: gp.cputime()  # random output
0.008000000000000000
sage: gp.factor('2^157-1')
[852133201, 1; 60726444167, 1; 1654058017289, 1; 2134387368610417, 1]
sage: 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'

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')
    28 # 32-bit
    38 # 64-bit

g_get_precision()
Return the current PARI precision for real number computations.

EXAMPLES:

    sage: gp.get_precision()
    28 # 32-bit
    38 # 64-bit

g_get_real_precision()
Return the current PARI precision for real number computations.

EXAMPLES:

    sage: gp.get_precision()
    28 # 32-bit
    38 # 64-bit

g_get_series_precision()
Return the current PARI power series precision.

EXAMPLES:
sage: gp.get_series_precision()
16

help(command)
Returns GP's help for command.

EXAMPLES:

sage: 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'

new_with_bits_prec(s, precision=0)
Creates 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: pi_def = gp(pi); pi_def
3.141592653589793238462643383 # 32-bit
3.1415926535897932384626433832795028842 # 64-bit
sage: pi_def.precision()
28 # 32-bit
38 # 64-bit
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
28 # 32-bit
38 # 64-bit
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()
28 # 32-bit
38 # 64-bit
**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:**

```python
sage: gp.set('x', '2')
sage: 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:**

```python
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')
28     # 32-bit
38     # 64-bit
```

**set_precision**(prec)
Sets 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:**

```python
sage: old_prec = gp.set_precision(53); old_prec
28     # 32-bit
38     # 64-bit
sage: gp.get_precision()
57
sage: gp.set_precision(old_prec)
57
sage: gp.get_precision()
28     # 32-bit
38     # 64-bit
```

**set_real_precision**(prec)
Sets 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:

```python
sage: old_prec = gp.set_precision(53); old_prec
28     # 32-bit
38     # 64-bit
sage: gp.get_precision()
57
sage: gp.set_precision(old_prec)
57
sage: gp.get_precision()
28     # 32-bit
38     # 64-bit
```

`set_seed(seed=None)`
Set the seed for gp interpreter.
The seed should be an integer.

EXAMPLES:

```python
sage: g = Gp()
sage: g.set_seed(1)
1
sage: [g.random() for i in range(5)]
[1546275796, 879788114, 1745191708, 771966234, 1247963869]
```

`set_series_precision(prec=None)`
Sets the PARI power series precision, and returns the old precision.

EXAMPLES:

```python
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
```

`version()`
Returns the version of GP being used.

EXAMPLES:

```python
sage: 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: `sage.interfaces.expect.ExpectElement`

EXAMPLES: This example illustrates dumping and loading GP elements to compressed strings.
sage: a = gp(39393)
sage: 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
1.047197551196597746154214461 # 32-bit
1.0471975511965977461542144610931676281 # 64-bit
sage: loads(dumps(x))
1.047197551196597746154214461 # 32-bit
1.0471975511965977461542144610931676281 # 64-bit

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

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)

sage.interfaces.gp gp_version()
EXAMPLES:

sage: gp.version() # not tested
((2, 4, 3), 'GP/PARI CALCULATOR Version 2.4.3 (development svn-12577)')

sage.interfaces.gp is_GpElement(x)
Returns True of x is a GpElement.

EXAMPLES:

sage: from sage.interfaces.gp import is_GpElement
sage: is_GpElement(gp(2))
(continues on next page)
True

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

```python
sage.interfaces.gp.reduce_load_GP()
```

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

**EXAMPLES:**

```python
sage: from sage.interfaces.gp import reduce_load_GP
sage: reduce_load_GP()
PARI/GP interpreter
```
INTERFACE FOR EXTRACTING DATA AND GENERATING IMAGES FROM JMOL READABLE FILES.

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

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

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

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

```python
def export_image(self, targetfile, datafile, datafile_cmd='script', image_type='PNG', figsize=5, **kwds)
```

This executes JmolData.jar to make an image file.

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

**OUTPUT:**
Image file, .png, .gif or .jpg (default .png)

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

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

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

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

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

```python
sage: from sage.interfaces.jmoldata import JmolData
sage: JData = JmolData()
sage: D = dodecahedron()
sage: from sage.misc.misc import SAGE_TMP
sage: archive_name = os.path.join(SAGE_TMP, "archive.jmol.zip")
...:
    import sys
    if sys.platform == 'cygwin':
        ....: import cygwin
        ....: archive_native = cygwin.cygpath(archive_native, 'w')
sage: script = 'set defaultdirectory "$0"

script SCRIPT'.format(archive_˓→native)
sage: testfile = os.path.join(SAGE_TMP, "testimage.png")
sage: JData.export_image(targetfile=testfile, datafile=script, image_type="PNG")
# optional -- java
sage: print(os.path.exists(testfile)) # optional -- java
True
```

`is_jvm_available()`

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

EXAMPLES:

Check that it returns a boolean:

```python
sage: from sage.interfaces.jmoldata import JmolData
sage: JData = JmolData()
sage: type(JData.is_jvm_available())
<... 'bool'>
```
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](http://page.math.tu-berlin.de/~kant/kash.html)

**Todo:** Update the following sentence.

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

The KASH interface offers three pieces of functionality:

1. `kash_console()` - A function that dumps you into an interactive command-line KASH session. Alternatively, type `!kash` from the Sage prompt.

2. `kash(expr)` - Creation of a Sage object that wraps a KASH object. This provides a Pythonic interface to KASH. For example, if `f=kash.new(10)`, then `f.Factors()` returns the prime factorization of 10 computed using KASH.

3. `kash.function_name(args ...)` - Call the indicated KASH function with the given arguments and return the result as a KASH object.

4. `kash.eval(expr)` - Evaluation of arbitrary KASH expressions, with the result returned as a string.

### 16.1 Issues

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

### 16.2 Tutorial

The examples in this tutorial require that `kash` be installed.
16.2.1 Basics

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

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

Next we obtain the result as a new KASH object.

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

We can do arithmetic and call functions on KASH objects:

```
sage: a*a  # optional -- kash
484
sage: a.Factorial()  # optional -- kash
1124000727777607680000
```

16.2.2 Integrated Help

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

16.2.3 Arithmetic

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

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

16.2.4 Variable assignment

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

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

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

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

Use `a.name()` to get the name of the KASH variable:
16.2.5 Integers and Rationals

We illustrate arithmetic with integers and rationals in KASH.

```
sage: F = kash.Factorization(4352)  # optional -- kash
sage: F[1]                        # optional -- kash
<2, 8>
<17, 1>
sage: F                          # optional -- kash
[<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: kash.GCD(15,25)             # optional -- kash
5
sage: kash.LCM(15,25)            # optional -- kash
75
sage: kash.Div(25,15)            # optional -- kash
1
sage: kash(17) % kash(5)         # optional -- kash
2
sage: kash.IsPrime(10007)        # optional -- kash
TRUE
FALSE
sage: kash.NextPrime(10007)      # optional -- kash
10009
```

16.2.6 Real and Complex Numbers

```
sage: kash.Precision()           # optional -- kash
30
sage: kash('R')                  # optional -- kash
Real field of precision 30
sage: kash.Precision(40)         # optional -- kash
40
sage: kash('R')                  # optional -- kash
Real field of precision 40
```

(continues on next page)
sage: z = kash('1 + 2*I')  # optional -- kash
1.000000000000000000000000000000000000000 + 2.000000000000000000000000000000000000000*I
sage: z**z  # optional -- kash
-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 + 0.92729521800161223242851262922*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
sage: x_c = kash('3+I')  # optional -- kash
sage: x_c.Argument()  # optional -- kash
0.321750554396642193401404614359
sage: x_c.Imaginary()  # optional -- kash
1.00000000000000000000000000000

16.2.7 Lists

Note that list appends are completely different in KASH than in Python. Use underscore after the function name for the mutation version.

sage: v = kash([1,2,3]); v  # optional -- kash
[ 1, 2, 3 ]
sage: v[1]  # optional -- kash
1
3
sage: v.Append([5])  # optional -- kash
[ 1, 2, 3, 5 ]
sage: v  # optional -- kash
\begin{verbatim}
[ 1, 2, 3 ]
sage: v.Append_([5, 6])  # optional -- kash
SUCCESS
sage: v  # optional -- kash
[ 1, 2, 3, 5, 6 ]
sage: v.Add(5)  # optional -- kash
[ 1, 2, 3, 5, 6, 5 ]
sage: v  # optional -- kash
[ 1, 2, 3, 5, 6 ]
sage: v.Add_(5)  # optional -- kash
SUCCESS
sage: v  # optional -- kash
[ 1, 2, 3, 5, 6, 5 ]
\end{verbatim}

The \texttt{Apply} command applies a function to each element of a list:

\begin{verbatim}
sage: L = kash([1,2,3,4])  # optional -- kash
sage: L.Apply('i -> 3*i')  # optional -- kash
[ 3, 6, 9, 12 ]
sage: L  # optional -- kash
[ 1, 2, 3, 4 ]
sage: L.Apply('IsEven')  # optional -- kash
[ FALSE, TRUE, FALSE, TRUE ]
sage: L  # optional -- kash
[ 1, 2, 3, 4 ]
\end{verbatim}

16.2.8 Ranges

the following are examples of ranges.

\begin{verbatim}
sage: L = kash([1..10])  # optional -- kash
sage: L  # optional -- kash
[ 1 .. 10 ]
sage: L = kash([2,4..100])  # optional -- kash
sage: L  # optional -- kash
[ 2, 4 .. 100 ]
\end{verbatim}

16.2.9 Sequences

16.2.10 Tuples

16.2.11 Polynomials

\begin{verbatim}
sage: f = kash('X^3 + X + 1')  # optional -- kash
sage: f + f  # optional -- kash
2*X^3 + 2*X + 2
sage: f * f  # optional -- kash
X^6 + 2*X^4 + 2*X^3 + X^2 + 2*X + 1
sage: f.Evaluate(10)  # optional -- kash
\end{verbatim}
16.2.12 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
Equation Order with defining polynomial X^5 + 4*X^4 - 56*X^2 - 16*X + 192 over Z
```

```
sage: f = kash('X^5 + 4*X^4 - 56*X^2 - 16*X + 192')  # optional -- kash
sage: O = f.EquationOrder()  # optional -- kash
sage: a = O.gen(2)  # optional -- kash
sage: a  # optional -- kash
[0, 1, 0, 0, 0]
```

```
sage: O.Basis()  # output somewhat random; optional -- kash
[  
  _NG.1,  
  _NG.2,  
  _NG.3,  
  _NG.4,  
  _NG.5  
]
sage: O.Discriminant()  # optional -- kash
1364202618880
sage: O.MaximalOrder()  # name sage2 below somewhat random; optional -- kash
Maximal Order of sage2
```

```
sage: 0 = kash('X^3 - 77')  # optional -- kash
sage: I = O.Ideal([5, 2, 1, 0])  # optional -- kash
sage: I
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>]
```

Determining whether an ideal is principal.
Computation of class groups and unit groups:

```python
sage: f = kash('X^5 + 4*X^4 - 56*X^2 -16*X + 192')  # optional -- kash
sage: O = kash.EquationOrder(f)                      # optional -- kash
sage: OK = O.MaximalOrder()                         # optional -- kash
sage: OK.ClassGroup()                               # name sage32 below random; optional -- kash
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
```

```python
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
```

```python
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] ]
```

### 16.2.13 Function Fields

```python
sage: k = kash.FiniteField(25)                      # optional -- kash
sage: kT = k.RationalFunctionField()               # optional -- kash
sage: kTy = kT.PolynomialAlgebra()                 # optional -- kash
sage: T = kT.gen(1)                                # optional -- kash
sage: y = kTy.gen(1)                               # optional -- kash
sage: f = y**3 + T**4 + 1                          # optional -- kash
```

### 16.3 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.

```python
sage: a = kash(range(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: sage.interfaces.expect.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.

custom()

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 - bool (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
1.41421356237309504880168872421

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>
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.

type

version()
```

```python
class sage.interfaces.kash.KashDocumentation
    Bases: list

class sage.interfaces.kash.KashElement(parent, value, is_name=False, name=None)
    Bases: sage.interfaces.expect.ExpectElement

sage.interfaces.kash.is_KashElement(x)
sage.interfaces.kash.kash_console()
sage.interfaces.kash.kash_version()
sage.interfaces.kash.reduce_load_Kash()
```
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 – A list of morphisms of chain complexes

OUTPUT:

• A KenzoSpectralSequence

EXAMPLES:

```
sage: from sage.interfaces.kenzo import BicomplexSpectralSequence # 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, 0, 2])})
sage: l = [M1, M2]
sage: E = BicomplexSpectralSequence(l) # optional - kenzo
e. group(2,0,1) # optional - kenzo
Additive abelian group isomorphic to Z/2 + Z
sage: E.table(3,0,2,0,2) # optional - kenzo
| 0 0 0 |
| Z/2 + Z/4 0 0 |
| 0 0 Z |
sage: E.matrix(2,2,0) # optional - kenzo
[ 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:

```python
sage: from sage.interfaces.kenzo import EilenbergMacLaneSpace # optional - kenzo
sage: e3 = EilenbergMacLaneSpace(ZZ, 3) # optional - kenzo
sage: [e3.homology(i) for i in range(8)] # optional - kenzo
[Z, 0, 0, ZZ, 0, C2, 0, C3]
sage: f3 = EilenbergMacLaneSpace(AdditiveAbelianGroup([2]), 3) # optional - kenzo
sage: [f3.homology(i) for i in range(8)] # optional - kenzo
[Z, 0, 0, Z, 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:

```python
sage: from sage.topology.simplicial_set import AbstractSimplex
sage: from sage.interfaces.kenzo import KAbstractSimplex,
    ...: KAbstractSimplex,

sage: SAbSm = AbstractSimplex(1, (2,0,3,2,1), name = 'SAbSm') # optional - kenzo
sage: KAbSm = KAbstractSimplex(SAbSm) # optional - kenzo
sage: SAbSm2 = SAbstractSimplex(KAbSm, 1) # optional - kenzo
sage: SAbSm.degeneracies() == SAbSm2.degeneracies() # optional - kenzo
True
sage: SAbSm.dimension() == SAbSm2.dimension() # optional - kenzo
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:

```python
sage: from sage.interfaces.kenzo import KChainComplex # optional - kenzo
sage: m1 = matrix(ZZ, 3, 2, [-1, 1, 3, -4, 5, 6])
sage: m4 = matrix(ZZ, 2, 2, [1, 2, 3, 6])
sage: m5 = matrix(ZZ, 2, 3, [2, 2, 2, -1, -1, -1])
sage: source_chcm = ChainComplex({1: m1, 4: m4, 5: m5}, degree = -1) # optional - kenzo
```
sage: kenzo_chcm = KChainComplex(sage_chcm) # optional - kenzo
sage: kenzo_chcm
[Kenzo Chain-Complex]
sage: kenzo_chcm.homology(5) # optional - kenzo
\[ \mathbb{Z} \times \mathbb{Z} \]

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: from sage.interfaces.kenzo import KChainComplexMorphism # optional - kenzo
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) # optional - kenzo
sage: g
[Kenzo Morphism (degree 0): KenzoChainComplex -> KenzoChainComplex]
sage: g.source_complex() # optional - kenzo
[Kenzo Chain-Complex]
sage: g.target_complex() # optional - kenzo
[Kenzo 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: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: from sage.interfaces.kenzo import KFiniteSimplicialSet # optional - kenzo
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)  # optional - kenzo
sage: KTriangle.homology(1)  # optional - kenzo
Z
sage: KTriangle.basis(1)  # optional - kenzo
['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))  # optional - kenzo
sage: KS1vS3.homology(3)  # optional - kenzo
Z

class sage.interfaces.kenzo.KenzoChainComplex(kenzo_object)
Bases: sage.interfaces.kenzo.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 – An integer number

OUTPUT:
• A list of the form ['G"dim"G0', 'G"dim"G1', 'G"dim"G2', ...].

EXAMPLES:

sage: from sage.interfaces.kenzo import KChainComplex  # optional - kenzo
sage: m1 = matrix(ZZ, 3, 2, [-1, 1, 3, -4, 5, 6])
  # optional - kenzo
sage: m4 = matrix(ZZ, 2, 2, [1, 2, 3, 6])
  # optional - kenzo
sage: m5 = matrix(ZZ, 2, 3, [2, 2, 2, -1, -1, -1])
  # optional - kenzo
sage: sage_chcm = ChainComplex({1: m1, 4: m4, 5: m5}, degree = -1)  # optional - kenzo
sage: kenzo_chcm = KChainComplex(sage_chcm)  # optional - kenzo
sage: kenzo_chcm
[K... Chain-Complex]
for i in range(6):
    Basis in dimension {i: %s} % (i, kenzo_chcm.basis(i)))  # optional - kenzo
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** – An integer number or None (default)
- **comb** – A 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:**

```python
sage: from sage.interfaces.kenzo import KChainComplex ...
```

**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:

- An homology group.

EXAMPLES:

```python
sage: from sage.interfaces.kenzo import Sphere  # optional - kenzo
sage: s2 = Sphere(2)  # optional - kenzo
sage: s2  # optional - kenzo
[\text{K1 Simplicial-Set}]
sage: s2.homology(2)  # optional - kenzo
\mathbb{Z}
```

`identity_morphism()`

Return the identity morphism (degree 0) between `self` and itself.

OUTPUT:

- A `KenzoChainComplexMorphism`

EXAMPLES:

```python
sage: from sage.interfaces.kenzo import Sphere  # optional - kenzo
sage: s2 = Sphere(2)  # optional - kenzo
sage: tp = s2.tensor_product(s2)  # optional - kenzo
sage: idnt = tp.identity_morphism()  # optional - kenzo
sage: type(idnt)  # optional - kenzo
<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` – An integer number 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:

```python
sage: from sage.interfaces.kenzo import Sphere  # optional - kenzo
sage: s2 = Sphere(2)  # optional - kenzo
sage: s3 = Sphere(3)  # optional - kenzo
(continues on next page)```
Sage 9.5 Reference Manual: Interpreter Interfaces, Release 9.5

(continued from previous page)

\begin{verbatim}
sage: tp22 = s2.tensor_product(s2) # optional - kenzo
sage: tp22 # optional - kenzo
[... Chain-Complex]
sage: tp23 = s2.tensor_product(s3) # optional - kenzo
sage: tp23 # optional - kenzo
[... Chain-Complex]
sage: null1 = tp22.null_morphism() # optional - kenzo
sage: null1 # optional - kenzo
[... Morphism (degree 0): K... -> K...]
sage: null2 = tp22.null_morphism(target = tp23, degree = -3) # optional - kenzo
sage: null2 # optional - kenzo
[... Morphism (degree -3): K... -> K...]
\end{verbatim}

\textbf{orgn()}

Return the :orgn slot of Kenzo, which stores as a list the origin of the object

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.interfaces.kenzo import Sphere # optional - kenzo
dsage: s2 = Sphere(2) # optional - kenzo
dsage: l2 = s2.loop_space() # optional - kenzo
sage: l2.orgn() # optional - kenzo
'(LOOP-SPACE [K... Simplicial-Set])'
sage: A = l2.cartesian_product(s2) # optional - kenzo
sage: A.orgn() # optional - kenzo
'(CRTS-PRDC [K... Simplicial-Group] [K... Simplicial-Set])'
\end{verbatim}

\textbf{tensor_product(other)}

Return the tensor product of self and other.

\textbf{INPUT:}

- other – The Kenzo object with which to compute the tensor product

\textbf{OUTPUT:}

- A \texttt{KenzoChainComplex}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.interfaces.kenzo import Sphere # optional - kenzo
dsage: s2 = Sphere(2) # optional - kenzo
dsage: s3 = Sphere(3) # optional - kenzo
dsage: p = s2.tensor_product(s3) # optional - kenzo
tsage: type(p) # optional - kenzo
<class 'sage.interfaces.kenzo.KenzoChainComplex'>
tsage: [p.homology(i) for i in range(8)] # optional - kenzo
[Z, 0, Z, Z, 0, Z, 0, 0]
\end{verbatim}
class sage.interfaces.kenzo.KenzoChainComplexMorphism(kenzo_object)

Bases: sage.interfaces.kenzo.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 (:intrslotinKenzo) and the strategy (:strtslotinKenzo). 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: from sage.interfaces.kenzo import Sphere, KenzoChainComplex # optional - kenzo
sage: from sage.libs.ecl import ecl_eval
sage: ZCC = KenzoChainComplex(ecl_eval("(z-chcm)")) # optional - kenzo
sage: ZCC
[K... Chain-Complex]
sage: s2 = Sphere(2) # optional - kenzo
sage: s3 = Sphere(3) # optional - kenzo
sage: tp = s2.tensor_product(s3) # optional - kenzo
sage: null = ZCC.null_morphism(tp) # optional - kenzo
sage: null
[K... Morphism (degree 0): K... -> K...]
sage: null.source_complex() # optional - kenzo
[K... Chain-Complex]
sage: null2 = null.change_source_target_complex(source = tp) # optional - kenzo
sage: null2
[K... Morphism (degree 0): K... -> K...]
sage: null2.source_complex() # optional - kenzo
[K... Filtered-Chain-Complex]
```
composite\(\text{object}=\text{None}\)

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:**

```python
sage: from sage.interfaces.kenzo import Sphere    # optional - kenzo
sage: s2 = Sphere(2)                               # optional - kenzo
sage: s3 = Sphere(3)                               # optional - kenzo
sage: tp22 = s2.tensor_product(s2)                  # optional - kenzo
sage: tp23 = s2.tensor_product(s3)                  # optional - kenzo
sage: idnt = tp22.identity_morphism()              # optional - kenzo
sage: idnt                                        # optional - kenzo
[K... Morphism (degree 0): K... -> K...]
```

```python
sage: null = tp23.null_morphism(target = tp22, degree = 4) # optional - kenzo
sage: null                                           # optional - kenzo
[K... Morphism (degree 4): K... -> K...]
```

```python
sage: idnt.composite((tp22, null))                 # optional - kenzo
[K... Morphism (degree 3): K... -> K...]
```

degree()

Return the degree of the morphism.

**OUTPUT:**

- An integer number, the degree of the morphism.

**EXAMPLES:**

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

(continues on next page)
sage: kenzo_chcm = KChainComplex(sage_chcm)  # optional
˓→ kenzo
sage: kenzo_chcm  # optional
˓→ kenzo
[K... Chain-Complex]
sage: differential_morphism = kenzo_chcm.differential()  # optional
˓→ kenzo
sage: differential_morphism  # optional
˓→ kenzo
[K... Morphism (degree -1): K... -> K...]
sage: differential_morphism.degree()  # optional
˓→ kenzo
-1
sage: differential_morphism.composite(differential_morphism).degree()  # optional
˓→optional - kenzo
-2
sage: kenzo_chcm.null_morphism().degree()  # optional
˓→ kenzo
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 omitted, the corresponding slot is inherited from self.

EXAMPLES:

sage: from sage.interfaces.kenzo import Sphere, KenzoChainComplex # optional
˓→ kenzo
sage: from sage.libs.ecl import ecl_eval
˓→ kenzo
sage: ZCC = KenzoChainComplex(ecl_eval("(z-chcm)"))  # optional
˓→ kenzo
sage: ZCC  # optional
˓→ kenzo
[K... Chain-Complex]
sage: s2 = Sphere(2)  # optional
˓→ kenzo
sage: s3 = Sphere(3)  # optional
˓→ kenzo
sage: tp = s2.tensor_product(s3)  # optional
˓→ kenzo
sage: tp  # optional
˓→ kenzo
[K... Filtered-Chain-Complex]
evaluation(dim, comb)

Apply the morphism on a combination comb of dimension dim.

INPUT:

• dim – An integer number

• comb – A list representing a formal sum of generators in the module of dimension dim. For example, to represent $G_7G_{12} + 3*G_7G_0 - 5*G_7G_3$ we use the list $[3, 'G_7G_0', -5, 'G_7G_3', 1, 'G_7G_{12}]$. Note that the generators must be in ascending order respect to the number after the second G in their representation; the parameter comb $= [1, 'G_7G_{12}', 3, 'G_7G_0', -5, 'G_7G_3']$ 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:

```python
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: differential_morphism = kenzo_chcm.differential()
```
opposite()  
Return the opposite morphism of self, i.e., -1 x self.

OUTPUT:

• A KenzoChainComplexMorphism

EXAMPLES:
source_complex()

Return the source chain complex of the morphism.

OUTPUT:

• A KenzoChainComplex

EXAMPLES:

```python
sage: from sage.interfaces.kenzo import KChainComplex  # optional
   -> kenzo
sage: m1 = matrix(ZZ, 3, 2, [-1, 1, 3, -4, 5, 6])
sage: m4 = matrix(ZZ, 2, 2, [1, 2, 3, 6])
sage: m5 = matrix(ZZ, 2, 3, [2, 2, 2, -1, -1, -1])
sage: sage_chcm = ChainComplex({1: m1, 4: m4, 5: m5}, degree = -1)  # optional
   -> kenzo
sage: kenzo_chcm = KChainComplex(sage_chcm)  # optional
   -> kenzo
sage: kenzo_chcm
   [K... Chain-Complex]
sage: idnt = kenzo_chcm.identity_morphism()  # optional
   -> kenzo
sage: idnt
   [K... Morphism (degree 0): K... -> K...]
sage: opps_id = idnt.opposite()  # optional
   -> kenzo
sage: opps_id
   [K... Morphism (degree 0): K... -> K...]
sage: kenzo_chcm.basis(4)  # optional
   -> kenzo
['G4G0', 'G4G1']
sage: idnt.evaluation(4, [2, 'G4G0', -5, 'G4G1'])  # optional
   -> kenzo
<2 * 4G0>
<-5 * 4G1>
```

```
source_complex()

Return the source chain complex of the morphism.

OUTPUT:

• A KenzoChainComplex

EXAMPLES:

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

```python
sage: sage_chcm = ChainComplex({1: m1, 4: m4, 5: m5}, degree = -1)  # optional
  → kenzo
```

```python
sage: kenzo_chcm = KChainComplex(sage_chcm)  # optional
  → kenzo
```

```python
sage: kenzo_chcm
  → kenzo
```

```
[K... Chain-Complex]
```

```python
sage: differential_morphism = kenzo_chcm.differential()  # optional
  → kenzo
```

```python
sage: differential_morphism
  → kenzo
```

```
[K... Morphism (degree -1): K... -> K...]
```

```python
sage: differential_morphism.source_complex()  # optional
  → kenzo
```

```
[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:**

```python
sage: from sage.interfaces.kenzo import KChainComplex  # optional
  → kenzo
```

```python
sage: m1 = matrix(ZZ, 3, 2, [-1, 1, 3, -4, 5, 6])
```

```python
sage: m4 = matrix(ZZ, 2, 2, [1, 2, 3, 6])
```

```python
sage: m5 = matrix(ZZ, 2, 3, [2, 2, 2, -1, -1, -1])
```

```python
sage: sage_chcm = ChainComplex({1: m1, 4: m4, 5: m5}, degree = -1)  # optional
  → kenzo
```

```python
sage: kenzo_chcm = KChainComplex(sage_chcm)  # optional
  → kenzo
```

```python
sage: kenzo_chcm
  → kenzo
```

```
[K... Chain-Complex]
```

```python
sage: idnt = kenzo_chcm.identity_morphism()  # optional
  → kenzo
```

```python
sage: idnt
  → kenzo
```

```
[K... Morphism (degree 0): K... -> K...]
```

```python
sage: opps_id = idnt.opposite()  # optional
  → kenzo
```

```python
sage: opps_id
  → kenzo
```

```
[K... Morphism (degree 0): K... -> K...]
```

(continues on next page)
sage: null = kenzo_chcm.null_morphism()  # optional
˓→ kenzo
sage: null  # optional
˓→ kenzo
[K... Morphism (degree 0): K... -> K...]
sage: idx2 = idnt.subtract(opps_id)  # optional
˓→ kenzo
sage: opps_idx2 = idx2.subtract\(...)  # optional
˓→ kenzo
sage: kenzo_chcm.basis(4)  # optional
˓→ kenzo
[‘G4G0’, ‘G4G1’]
˓→ kenzo
<4 * G4G0>
<-10 * G4G1>
˓→ kenzo
<4 * G4G0>
<-10 * G4G1>
---

**sum**(object=None)

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

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

(continues on next page)
target_complex()

Return the target chain complex of the morphism.

OUTPUT:

• A KenzoChainComplex

EXAMPLES:

```python
sage: from sage.interfaces.kenzo import KChainComplex       # optional
    kenzo
sage: m1 = matrix(ZZ, 3, 2, [-1, 1, 3, -4, 5, 6])
sage: m4 = matrix(ZZ, 2, 2, [1, 2, 3, 6])
sage: m5 = matrix(ZZ, 2, 3, [2, 2, 2, -1, -1, -1])
```
sage: sage_chcm = ChainComplex({1: m1, 4: m4, 5: m5}, degree = -1)  # optional
    — kenzo
sage: kenzo_chcm = KChainComplex(sage_chcm)  # optional
    — kenzo
sage: kenzo_chcm  # optional
    — kenzo
[K... Chain-Complex]
sage: differential_morphism = kenzo_chcm.differential()  # optional
    — kenzo
sage: differential_morphism  # optional
    — kenzo
[K... Morphism (degree -1): K... -> K...]
sage: differential_morphism.target_complex()  # optional
    — kenzo
[K... Chain-Complex]

class sage.interfaces.kenzo.KenzoObject(kenzo_object)
    Bases: sage.structure.sage_object.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: sage.interfaces.kenzo.KenzoSimplicialSet

Wrapper around Kenzo simplicial groups.

classifying_space()
    Return the classifying space.

OUTPUT:

• A KenzoSimplicialGroup

EXAMPLES:

sage: from sage.interfaces.kenzo import MooreSpace  # optional - kenzo
sage: m2 = MooreSpace(2,4)  # optional - kenzo
sage: l2 = m2.loop_space()  # optional - kenzo
sage: c = l2.classifying_space()  # optional - kenzo
sage: type(c)  # optional - kenzo
<class 'sage.interfaces.kenzo.KenzoSimplicialGroup'>
sage: [c.homology(i) for i in range(8)]  # optional - kenzo
[Z, 0, 0, 0, C2, 0, 0, 0]

class sage.interfaces.kenzo.KenzoSimplicialSet(kenzo_object)
    Bases: sage.interfaces.kenzo.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:**

```python
sage: from sage.interfaces.kenzo import Sphere # optional - kenzo
sage: s2 = Sphere(2) # optional - kenzo
sage: s3 = Sphere(3) # optional - kenzo
sage: p = s2.cartesian_product(s3) # optional - kenzo
sage: type(p) # optional - kenzo
<class 'sage.interfaces.kenzo.KenzoSimplicialSet'>
sage: [p.homology(i) for i in range(6)] # optional - kenzo
[Z, 0, Z, Z, 0, Z]
```

**em_spectral_sequence()**

Return the Eilenberg-Moore spectral sequence of *self*.

**OUTPUT:**

- A *KenzoSpectralSequence*

**EXAMPLES:**

```python
sage: from sage.interfaces.kenzo import Sphere # optional - kenzo
sage: S2 = Sphere(2) # optional - kenzo
sage: EMS = S2.em_spectral_sequence() # optional - kenzo
sage: EMS.table(0, -2, 2, -2, 2) # optional - kenzo
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:**

```python
sage: from sage.interfaces.kenzo import Sphere # optional - kenzo
sage: s2 = Sphere(2) # optional - kenzo
sage: p = s2.cartesian_product(s2) # optional - kenzo
sage: p.homotopy_group(3) # optional - kenzo
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: from sage.interfaces.kenzo import Sphere # optional - kenzo
sage: s2 = Sphere(2) # optional - kenzo
sage: s3 = Sphere(3) # optional - kenzo
sage: j = s2.join(s3) # optional - kenzo
sage: type(j) # optional - kenzo
<class 'sage.interfaces.kenzo.KenzoSimplicialSet'>
sage: [j.homology(i) for i in range(6)] # optional - kenzo
[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: from sage.interfaces.kenzo import Sphere # optional - kenzo
sage: s2 = Sphere(2) # optional - kenzo
sage: l2 = s2.loop_space() # optional - kenzo
sage: type(l2) # optional - kenzo
<class 'sage.interfaces.kenzo.KenzoSimplicialGroup'>
sage: l2 = s2.loop_space() # optional - kenzo
sage: [l2.homology(i) for i in range(8)] # optional - kenzo

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: from sage.interfaces.kenzo import Sphere
sage: S2 = Sphere(2)  # optional - kenzo
sage: S3 = Sphere(3)  # optional - kenzo
sage: P = S2.cartesian_product(S3)  # optional - kenzo
sage: E = P.serre_spectral_sequence()  # optional - kenzo
sage: E.table(0, 0, 2, 0, 3)  # optional - kenzo
\begin{verbatim}
Z 0 Z
0 0 0
0 0 0
Z 0 Z
\end{verbatim}

Warning: This method assumes that the underlying space is simply connected. You might get wrong answers if it is not.

\subsubsection*{smash_product}(other)

Return the smash product of self and other.

\textbf{INPUT:}

- other – the Kenzo simplicial set with which the smash product is made

\textbf{OUTPUT:}

- A \texttt{KenzoSimplicialSet}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.interfaces.kenzo import Sphere  # optional - kenzo
sage: s2 = Sphere(2)  # optional - kenzo
sage: s3 = Sphere(3)  # optional - kenzo
sage: s = s2.smash_product(s3)  # optional - kenzo
sage: type(s)  # optional - kenzo
<class 'sage.interfaces.kenzo.KenzoSimplicialSet'>
sage: [s.homology(i) for i in range(6)]  # optional - kenzo
[Z, 0, 0, 0, 0, Z]
\end{verbatim}

\subsubsection*{suspension()}

Return the suspension of the simplicial set.

\textbf{OUTPUT:}

- A \texttt{KenzoSimplicialSet}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.interfaces.kenzo import EilenbergMacLaneSpace  # optional - kenzo
˓→kenzo
sage: e3 = EilenbergMacLaneSpace(ZZ, 3)  # optional - kenzo
˓→kenzo
sage: s = e3.suspension()  # optional - kenzo
˓→kenzo
sage: type(s)  # optional - kenzo
˓→kenzo
<class 'sage.interfaces.kenzo.KenzoSimplicialSet'>
\end{verbatim}
sage: [s.homology(i) for i in range(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: from sage.interfaces.kenzo import Sphere  # optional - kenzo
sage: S3 = Sphere(3)  # optional - kenzo
sage: E = S3.sw_spectral_sequence()  # optional - kenzo
sage: T = E.table(0, 0, 4, 0, 4)  # optional - kenzo
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

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: from sage.interfaces.kenzo import Sphere  # optional - kenzo
sage: s2 = Sphere(2)  # optional - kenzo
sage: s3 = Sphere(3)  # optional - kenzo
sage: w = s2.wedge(s3)  # optional - kenzo
sage: type(w)  # optional - kenzo
<class 'sage.interfaces.kenzo.KenzoSimplicialSet'>
sage: [w.homology(i) for i in range(6)]  # optional - kenzo
[Z, 0, Z, Z, 0, 0]

class sage.interfaces.kenzo.KenzoSpectralSequence(kenzo_object)

Bases: sage.interfaces.kenzo.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: from sage.interfaces.kenzo import Sphere  # optional - kenzo
sage: S3 = Sphere(3)  # optional - kenzo
sage: L = S3.loop_space()  # optional - kenzo
sage: EMS = L.em_spectral_sequence()  # optional - kenzo
sage: EMS.table(1,-5,-2,5,8)  # optional - kenzo
0  Z  Z + 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)  # optional - kenzo
[ 2 -2 2]
sage: EMS.differential(1, -3, 8)  # optional - kenzo
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: from sage.interfaces.kenzo import Sphere  # optional - kenzo
sage: S2 = Sphere(2)  # optional - kenzo
sage: EMS = S2.em_spectral_sequence()  # optional - kenzo
sage: EMS.group(0, -1, 2)  # optional - kenzo
Additive abelian group isomorphic to Z
sage: EMS.group(0, -1, 3)  # optional - kenzo
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: from sage.interfaces.kenzo import Sphere  # optional - kenzo
sage: S3 = Sphere(3)  # optional - kenzo
sage: L = S3.loop_space()  # optional - kenzo
sage: EMS = L.em_spectral_sequence()  # optional - kenzo
sage: EMS.table(1, -5, -2, 5, 8)  # optional - kenzo
0  Z  Z + Z + Z  Z + Z + Z
0  0  0  0
0  0  Z  Z + Z
0  0  0  0
```
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:**

```python
sage: from sage.interfaces.kenzo import Sphere # optional - kenzo
sage: S2 = Sphere(2) # optional - kenzo
sage: EMS = S2.em_spectral_sequence() # optional - kenzo
sage: EMS.table(0, -2, 2, -2, 2) # optional - kenzo
```

```
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:**

- \(m\) – A positive integer. The order of the nontrivial homology group.
- \(n\) – The dimension in which the homology is not trivial

**OUTPUT:**

- A KenzoSimplicialSet

**EXAMPLES:**

```python
sage: from sage.interfaces.kenzo import MooreSpace # optional - kenzo
sage: m24 = MooreSpace(2,4) # optional - kenzo
sage: m24 # optional - kenzo
```

```
[K10 Simplicial-Set]
```

```python
sage: [m24.homology(i) for i in range(8)] # optional - kenzo
```

```
[Z, 0, 0, 0, C2, 0, 0, 0]
```
Sage 9.5 Reference Manual: Interpreter Interfaces, Release 9.5

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: 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

#
#
#
#
#

optional
optional
optional
optional
optional

-

kenzo
kenzo
kenzo
kenzo
kenzo

# optional - kenzo
# optional - kenzo

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 – An integer number (optional, default 0)
• end – An integer number greater than or equal to start (optional, default 15)
OUTPUT:
• A ChainComplex
EXAMPLES:
sage: from sage.interfaces.kenzo import KChainComplex, SChainComplex
˓→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)
˓→kenzo
sage: SChainComplex(KChainComplex(sage_chcm)) == sage_chcm
˓→kenzo
True
sage:
sage:
sage:
sage:

from sage.interfaces.kenzo import SChainComplex, Sphere
S4 = Sphere(4)
# optional - kenzo
C = SChainComplex(S4)
# optional - kenzo
C
# optional - kenzo

# optional -␣

# optional -␣
# optional -␣

# optional - kenzo

(continues on next page)

126

Chapter 17. Library interface to Kenzo


Chain complex with at most 3 nonzero terms over Integer Ring

```
sage: C._ascii_art_()  # optional - kenzo
0 <-- C_4 <-- 0 ... 0 <-- C_0 <-- 0
sage: [C.homology(i) for i in range(6)]  # optional - kenzo
[Z, 0, 0, 0, Z, 0]
```

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

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

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

**OUTPUT:**
- A finite SimplicialSet.

**EXAMPLES:**

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

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

Return the n dimensional sphere as a Kenzo simplicial set.

**INPUT:**
- `n` – the dimension of the sphere

**OUTPUT:**
- A `KenzoSimplicialSet`

**EXAMPLES:**
sage.interfaces.kenzo.build_morphism(source_complex, target_complex, degree, algorithm, strategy, orgn)

Build a morphism of chain complexes by means of the corresponding build-mrph Kenzo function.

INPUT:

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

OUTPUT:

- A KenzoChainComplexMorphism

EXAMPLES:

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

sage.interfaces.kenzo.k2s_matrix(kmatrix)

Convert an array of ECL to a matrix of Sage.

INPUT:

- kmatrix – An array in ECL

EXAMPLES:

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

(continues on next page)
sage.interfaces.kenzo.morphism_dictmat
Computes a list of matrices in ECL associated to a morphism in Sage.

**INPUT:**
- morphism – A morphism of chain complexes

**OUTPUT:**
- A EclObject

**EXAMPLES:**

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

sage: morphism_dictmat(f)  # optional - kenzo
<ECL: ((2 . #2A()) (1 . #2A()) (0 . #2A((1 1)))))>
```

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

**INPUT:**
- slist – A list in Sage

**OUTPUT:**
- A EclObject

**EXAMPLES:**

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

sage: pairing(l)  # optional - kenzo
<ECL: ((2 . #2A()) (1 . #2A()) (0 . #2A((1 1)))))>
```

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

**INPUT:**
- sdictmat – A dictionary in Sage

**OUTPUT:**
- A EclObject

**EXAMPLES:**

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

sage: B = Matrix([[1,2],[2,1],[1,1]])

sage: d = {1 : A, 2 : B}
```

(continues on next page)
sage.interfaces.kenzo.s2k_listofmorphisms(l)

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

INPUT:

• l – A list of morphisms of chain complexes

OUTPUT:

• A EclObject

EXAMPLES:

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

sage.interfaces.kenzo.s2k_matrix(smatrix)

Convert a matrix of Sage to an array of ECL.

INPUT:

• smatrix – A matrix in Sage

OUTPUT:

• A EclObject

EXAMPLES:

```python
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))>
```
CALL TO THE PROGRAM COUNT FROM LATTÉ 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:**

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

Counting integer points from either the H or V representation:

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

Ehrhart polynomial:

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

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

```python
sage: opts = {'cdd': True,
          'multivariate_generating_function': True,
          'raw_output': True}
sage: cddin = P.cdd_Hrepresentation()  # optional - latte_int
```

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

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

Call to the function integrate from LattE integrale.

INPUT:

• arg – a cdd or LattE description string.

• polynomial – multivariate polynomial or valid LattE polynomial description string. If given, the valuation parameter of LattE is set to integrate, and is set to volume otherwise.

• algorithm – (default: ‘triangulate’) the integration method. Use ‘triangulate’ for polytope triangulation or ‘cone-decompose’ for tangent cone decomposition method.

• raw_output – if True then return directly the output string from LattE.

• verbose – if True then return directly verbose output from LattE.

• For all other options of the integrate program, consult the LattE manual.

OUTPUT:

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

EXAMPLES:

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

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

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

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

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

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

Polynomials given as a string in LattE description are also accepted:
sage: integrate(P.cdd_Hrepresentation(), '[[1,[2,2,2]]]', cdd=True)  # optional - ~latte_int
4096/27

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

INPUT:

• polynomial – a multivariate polynomial.

OUTPUT:

A string that describes the monomials list and exponent vectors.
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:**

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

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

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

sage: a4.Cartan()  # optional - lie
[[ 2, -1, 0, 0],
 [ -1, 2, -1, 0],
 [ 0, -1, 2, -1],
 [ 0, 0, -1, 2]]

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

135
19.1 Tutorial

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

You can perform basic arithmetic operations in LiE.

```python
sage: lie.eval('19+68') # optional - lie
     '87'
sage: a = lie('1111111111*1111111111') # optional - lie
     123456790987654321
sage: a/1111111111 # optional - lie
     1111111111
sage: a = lie('345') # optional - lie
sage: a^2+3*a-5 # optional - lie
     120055
sage: _ / 7*a # optional - lie
     5916750
```

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

```python
sage: v = lie('[3,2,6873,-38]') # optional - lie
     6873
sage: v*v # optional - lie
     47239586
sage: v+234786 # optional - lie
     [3,2,6873,-38,234786]
```

You can also work with matrices in LiE.

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

(continues on next page)
LiE handles multivariate (Laurent) polynomials.

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

```python
sage: lie.eval("f(int x) = 2*x") # optional - lie
''
```

```python
sage: lie.f(984) # optional - lie
1968
```

```python
sage: lie.eval("f(int n) = a=3*n-7; if a < 0 then a = -a fi; 7^a+a^3-4*a-57") # optional, -- lie
''
```

```python
sage: lie.f(2) # optional - lie
-53
```

```python
sage: lie.f(5) # optional - lie
5765224
```

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

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

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

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

Integers:

```python
sage: a = lie('1234') # optional - lie
sage: b = a.sage(); b # optional - lie
1234
```

```python
sage: type(b) # optional - lie
<class 'sage.rings.integer.Integer'>
```

Vectors:
**Matrices:**

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

**Polynomials:**

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

**Text:**

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

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

```python
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]') # optional - lie
sage: b1 = lie('[[1,2],[3,4]]') # optional - lie
sage: b2 = lie('[1,2][3,4]') # optional - lie
sage: eval_pol(a, b1) # optional - lie
4
sage: eval_pol(a, b2) # optional - lie
18
```

**AUTHORS:**

- Mike Hansen 2007-08-27
class sage.interfaces.lie.LiE(maxread=None, script_subdirectory=None, logfile=None, server=None)

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

Interface to the LiE interpreter.

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

console()

Spawn a new LiE command-line session.

EXAMPLES:

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

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

EXAMPLES:

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

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

EXAMPLES:

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

get_using_file(var)

EXAMPLES:

```
sage: lie.get_using_file('x')
Traceback (most recent call last):
... Not ImplementedError
```

help(command)

Return a string of the LiE help for command.

EXAMPLES:
read(filename)

EXAMPLES:

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

set(var, value)

Set the variable var to the given value.

EXAMPLES:

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

version()

EXAMPLES:

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

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

EXAMPLES:

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

sage.interfaces.lie.lie_version()

EXAMPLES:

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

sage.interfaces.lie.reduce_load_lie()

EXAMPLES:

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

```python
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('(+'%(a.name()) '(%(b.name())))')
'8'
```

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

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

AUTHORS:  
– William Stein (first version)  

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

   Bases: sage.interfaces.expect.Expect

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

console()
Spawn a new Lisp command-line session.

EXAMPLES:

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

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

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

function_call(function, args=None, kwds=None)
Calls the Lisp function with given args and kwds. For Lisp functions, the kwds are ignored.

EXAMPLES:

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

get(var)
EXAMPLES:

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

help(command)
EXAMPLES:

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

kill(var)
EXAMPLES:

```sage
sage: lisp.kill('x')
Traceback (most recent call last):
(continues on next page)```
```python
...  
NotImplementedError

**set**(var, value)
Set the variable var to the given value.

**EXAMPLES:**

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

**version()**
Returns the version of Lisp being used.

**EXAMPLES:**

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

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

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

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

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

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

**sage.interfaces.lisp.lisp_console()**
Spawn a new Lisp command-line session.

**EXAMPLES:**

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

**sage.interfaces.lisp.reduce_load_Lisp()**
**EXAMPLES:**
```python
sage: from sage.interfaces.lisp import reduce_load_Lisp
sage: reduce_load_Lisp()
Lisp Interpreter
```
Sage provides an interface to the Macaulay2 computational algebra system. This system provides extensive functionality for commutative algebra. You do not have to install any optional packages.

The Macaulay2 interface offers three pieces of functionality:

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

**EXAMPLES:**

```
sage: macaulay2('3/5 + 7/11')   # optional - macaulay2
68
--
55
sage: f = macaulay2('f = i -> i^3')   # optional - macaulay2
sage: f
# optional - macaulay2
f
sage: f(5)   # optional - macaulay2
125
sage: R = macaulay2('ZZ/5[x,y,z]')   # optional - macaulay2
sage: R
# optional - macaulay2
ZZ
--[x...z]
5
sage: x = macaulay2('x')   # optional - macaulay2
sage: y = macaulay2('y')   # optional - macaulay2
sage: (x+y)^5   # optional - macaulay2
5 5
x + y
sage: parent((x+y)^5)   # optional - macaulay2
Macaulay2
```
The name of the variable to which a Macaulay2 element is assigned internally can be passed as an argument. This is useful for types like polynomial rings which acquire that name in Macaulay2:

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

sage: f = macaulay2('x^4 + 2*x*y^3 + x*y*z*w + x*y*w^2'  # optional - macaulay2
                 ....:                     + 2*x*z*w^2 + y^4 + y^3*w + 2*y^2*z*w + z^4 + w^4')
sage: f                                     # optional - macaulay2
4 3 4 4 2 3 2 2 2 4
x + 2*x*y + y + z + x*y w + y w + x*y*z*w + 2y z*w + x*y*w + 2x*z*w + w

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

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

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

1 2
Matrix R <--- R

sage: macaulay2.eval('compactMatrixForm = true;') # optional - macaulay2
```

**AUTHORS:**

- Kiran Kedlaya and David Roe (2006-02-05, during Sage coding sprint)
- William Stein (2006-02-09): inclusion in Sage; prompt uses regexp, calling of Macaulay2 functions via `__call__`
- 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.
EXAMPLES:

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

```
console()
Spawn a new M2 command-line session.

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

```
cputime(t=\text{None})
EXAMPLES:

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

```
eval(code, strip=True, **kwds)
Send the code x to the Macaulay2 interpreter and return the output as a string suitable for input back into Macaulay2, if possible.

INPUT:
• code – str
• strip – ignored

EXAMPLES:

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

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

INPUT:
• var - string: the name of the variable in Macaulay2

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

EXAMPLES:

```
sage: macaulay2.set("a", "2")  # optional - macaulay2
sage: macaulay2.get("a")       # optional - macaulay2
2
```
Note that the following syntax is used to obtain a `Macaulay2Element` instead:

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

`help(s)`

**EXAMPLES:**

```python
sage: macaulay2.help("load")  # optional - macaulay2 - 1st call might be chatty.
→...
...
```

```python
sage: macaulay2.help("load")  # optional - macaulay2
load...
****...
...
* "input" -- read Macaulay2 commands and echo
* "notify" -- whether to notify the user when a file is loaded...
```

`ideal(*gens)`

Return the ideal generated by gens.

**INPUT:**

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

**OUTPUT:**

the Macaulay2 ideal generated by the given list of gens

**EXAMPLES:**

```python
sage: R2 = macaulay2.ring('QQ', '[x, y]'); R2  # optional - macaulay2
QQ[x...y]
```

```python
sage: I = macaulay2.ideal( ('y^2 - x^3', 'x - y') ); I  # optional - macaulay2
ideal (- x + y , x - y)
```

```python
sage: J = I^3; J.gb().gens().transpose()  # optional - macaulay2
{ -9 } | y9-3y8+3y7-y6 |
{ -7 } | xy6-2xy5+xy4-y7+2y6-y5 |
{ -5 } | x2y3-x2y2-2xy4+2xy3+y5-y4 |
{ -3 } | x3-3x2y+3xy2-y3 |
```

`new_from(type, value)`

Return a new `Macaulay2Element` of type `type` constructed from `value`.

**EXAMPLES:**

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

```python
sage: l  # optional - macaulay2
MutableList{...3...}
```

```python
sage: list(l)  # optional - macaulay2
[1, 2, 3]
```

`options(*get_value, **set_value)`

Global options for Macaulay2 elements.
OPTIONS:

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

EXAMPLES:

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

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

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

See GlobalOptions for more features of these options.

**restart()**
Restart Macaulay2 interpreter.

**ring**
(base_ring='ZZ', vars='[x]', order='Lex')
Create a Macaulay2 polynomial ring.

INPUT:

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

OUTPUT: a Macaulay2 ring

EXAMPLES:

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

```sage
sage: R1 = macaulay2.ring('ZZ/7', '[a..d]', 'GRevLex')  # optional - macaulay2
sage: R1.describe()  # optional - macaulay2
```

This is a polynomial ring over the rational numbers:

```sage
sage: R2 = macaulay2.ring('QQ', '[x, y]')  # optional - macaulay2
sage: R2.describe()  # optional - macaulay2
```

(continues on next page)
set(var, value)
Set the variable var to the given value.

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

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

set_seed(seed=None)
Set the seed for Macaulay2 interpreter.

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

OUTPUT: the new seed

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

use(R)
Use the Macaulay2 ring R.

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

version()
Returns the version of Macaulay2.

EXAMPLES:
```python
sage: macaulay2.version()    # optional - macaulay2
(1, 1...
class sage.interfaces.macaulay2.Macaulay2Element(parent, value, is_name=False, name=None)
Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.expect.ExpectElement

Instances of this class represent objects in Macaulay2.

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

```python
# _sage_() EXAMPLES:
sage: macaulay2(ZZ).sage()            # optional - macaulay2, indirect doctest
     Integer Ring
sage: macaulay2(QQ).sage()             # optional - macaulay2
     Rational Field
sage: macaulay2(2).sage()              # optional - macaulay2
     2
sage: macaulay2(1/2).sage()            # optional - macaulay2
     1/2
sage: macaulay2(2/1).sage()            # optional - macaulay2
     2
sage: _.parent()                       # optional - macaulay2
     Rational Field
sage: macaulay2([1,2,3]).sage()        # optional - macaulay2
     [1, 2, 3]
sage: m = matrix([[1,2],[3,4]])
sage: macaulay2(m).sage()              # optional - macaulay2
     [1 2]
     [3 4]
sage: D = macaulay2('hashTable {4 => 1, 2 => 3}')  # optional - macaulay2
sage: D.pairs()                         # optional - macaulay2
     {(4, 1), (2, 3)}
sage: D.sage() == {4: 1, 2: 3}          # optional - macaulay2
     True
sage: macaulay2(QQ['x,y']).sage()      # optional - macaulay2
     Multivariate Polynomial Ring in x, y over Rational Field
sage: macaulay2(QQ['x']).sage()        # optional - macaulay2
     Univariate Polynomial Ring in x over Rational Field
sage: macaulay2(GF(7)['x,y']).sage()   # optional - macaulay2
     Multivariate Polynomial Ring in x, y over Finite Field of size 7
sage: macaulay2(GF(7)).sage()          # optional - macaulay2
     Finite Field of size 7
sage: R.<x,y> = QQ[]
sage: macaulay2(x^2+y^2+1).sage()    # optional - macaulay2
     x^2 + y^2 + 1
sage: R = macaulay2("QQ[x,y]")        # optional - macaulay2
```

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

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

sage: S = ZZ['x,y'].quotient('x^2-y')
sage: macaulay2(S).sage() == S  # optional - macaulay2
True
sage: S = GF(101)['x,y'].quotient('x^2-y')
sage: macaulay2(S).sage() == S  # optional - macaulay2
True
sage: R = GF(13)['a,b']['c,d']
sage: macaulay2(R).sage() == R  # optional - macaulay2
True
sage: macaulay2('a^2 + c').sage() == R('a^2 + c')  # optional - macaulay2
True
sage: macaulay2.substitute('a', R).sage().parent() is R  # optional - macaulay2
True

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

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

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

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

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

Chain complexes and maps of chain complexes can be converted:
sage: R = ZZ['a,b,c']
sage: C = macaulay2(ideal(R.gens())).resolution()  # optional - macaulay2
sage: ascii_art(C.sage())  # optional - macaulay2

\[
\begin{array}{ccc}
-\mathbf{b} & 0 & -\mathbf{c} \\
\mathbf{a} & -\mathbf{c} & 0 \\
a & b & c \\
0 & 0 & b \\
\end{array}
\]

\[
0 \leftarrow C_0 \leftarrow C_1 \leftarrow C_2 \leftarrow C_3 \leftarrow 0
\]
sage: F = C.dot('dd')  # optional - macaulay2
sage: G = F.sage()  # optional - macaulay2
sage: G.in_degree(2)  # optional - macaulay2

\[
\begin{array}{ccc}
-\mathbf{b} & 0 & -\mathbf{c} \\
\mathbf{a} & -\mathbf{c} & 0 \\
a & b & c \\
0 & 0 & b \\
\end{array}
\]

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...
  To:  Chain complex with at most 4 nonzero terms over Multivariate Polynomial...

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

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

Elements of quotient rings:

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

\texttt{after\_print\_text}()

Obtain type information for this Macaulay2 element.

This is the text that is displayed using \texttt{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 \texttt{after\_print} in \texttt{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

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

EXAMPLES:

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

dot(x)
EXAMPLES:

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

external_string()
EXAMPLES:

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

name(new_name=None)
Get or change the name of this Macaulay2 element.

INPUT:

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

Note that this can overwrite existing variables in the system.

EXAMPLES:

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

The name can also be given at definition:
sage: A = macaulay2(ZZ['x,y,z'], name='A')  # optional - macaulay2
sage: A.name()  # optional - macaulay2
'A'

sage: A^1  # optional - macaulay2
A

sage_polystring()

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

EXAMPLES:

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

sharp(x)

EXAMPLES:

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

starstar(x)

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

EXAMPLES:

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

structure_sheaf()

EXAMPLES:

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

```
sage: X.sheaf()  # optional - macaulay2
```

```
sage: X.sheaf()  # optional - macaulay2
```
**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:**

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

**substitute(*args, **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:**

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

**to_sage(*args, **kwds)**

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

**underscore(x)**

**EXAMPLES:**

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

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

Bases: `sage.interfaces.expect.ExpectFunction`

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

Bases: `sage.interfaces.expect.FunctionElement`

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

**EXAMPLES:**

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

**sage.interfaces.macaulay2.macaulay2_console()**

Spawn a new M2 command-line session.

**EXAMPLES:**
sage.interfaces.macaulay2.reduce_load_macaulay2()
Used for reconstructing a copy of the Macaulay2 interpreter from a pickle.

EXAMPLES:

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

sage.interfaces.macaulay2.remove_output_labels(s)
Remove output labels of Macaulay2 from a string.

- s: output of Macaulay2
- s: string

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

Return type: string

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

EXAMPLES:

```
sage: from sage.interfaces.macaulay2 import remove_output_labels
sage: output = 'o1 = QQ \[x, y\]

o1 : PolynomialRing
' sage: remove_output_labels(output)
'QQ \[x, y\]\n
PolynomialRing
'
```
Sage provides an interface to the Magma computational algebra system. This system provides extensive functionality for number theory, group theory, combinatorics and algebra.

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

The Magma interface offers three pieces of functionality:

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

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

### 22.1 Parameters

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

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

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

We verify that an obviously principal ideal is principal:

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

22.3 Long Input

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

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

22.4 Garbage Collection

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

First, create a matrix `m` in Sage:

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

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

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

It is called `_sage_[...]' in Magma:

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

It's there:
22.5 Other Examples

We compute a space of modular forms with character.

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

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

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

We coerce some polynomial rings into Magma:

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

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

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

AUTHORS:

- William Stein (2005): initial version
- William Stein (2006-03-09): added nvals argument for magma.functions...

class `sage.interfaces.magma.Magma`

Bases: `sage.interfaces.tab_completion.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
1.1
sage: magma.SetDefaultRealFieldPrecision(30, nvals=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 - a string

EXAMPLES: Attaching a file that exists is fine:

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

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

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

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 - a string

EXAMPLES:

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

GetNthreads()
Get the number of threads used in Magma.

EXAMPLES:
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.setVerbose("Groebner", 2)       # optional - magma
sage: magma.getVerbose("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.setNthreads(2)                 #optional - magma
sage: magma.getNthreads()                 #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.setVerbose("Groebner", 2)       # optional - magma
sage: magma.getVerbose("Groebner")         # optional - magma
2
```

attach(filename)
Attach the given file to the running instance of Magma.

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

INPUT:

- filename - a string

EXAMPLES: Attaching a file that exists is fine:

```
sage: SAGE_EXTCODE = SAGE_ENV['SAGE_EXTCODE']                          # optional - magma
sage: magma.attach('%s/magma/sage/basic.m' %SAGE_EXTCODE)            # optional - magma
```
Attaching a file that doesn’t exist raises an exception:

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

**attach_spec** *(filename)*

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

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

**INPUT:**

- filename - a string

**EXAMPLES:**

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

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

This is a wrapper around the Magma constructor nameleft gens
returning nvals.

**INPUT:**

- left - something coercable 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:

```python
sage: V = magma.RModule(ZZ,3)  # optional - magma
sage: V  # optional - magma
RModule(IntegerRing(), 3)
sage: magma.bar_call(V, 'quo', [[1,2,3]], nvals=1)  # optional - magma
RModule(IntegerRing(), 2)
sage: magma.bar_call(V, 'quo', [[1,2,3]], nvals=2)  # optional - magma
(RModule(IntegerRing(), 2),
 Mapping from: RModule(IntegerRing(), 3) to RModule(IntegerRing(), 2))
sage: magma.bar_call(V, 'quo', V, nvals=2)  # optional - magma
(RModule(IntegerRing(), 0),
 Mapping from: RModule(IntegerRing(), 3) to RModule(IntegerRing(), 0))
```
**chdir**(dir)
Change the Magma interpreter’s current working directory.

**INPUT:**

- dir – a string

**EXAMPLES:**

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

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

**INPUT:**

- var - a string

**EXAMPLES:**

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

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

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

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

```
sage: magma = Magma()            # optional - magma
sage: a = magma('100')           # optional - magma
sage: a.name()                   # optional - magma
 '_sage_[1]'
sage: del a                     # optional - magma
sage: b = magma('257')           # optional - magma
sage: b.name()                   # optional - magma
 '_sage_[1]'
sage: del b                     # optional - magma
sage: magma('_sage_[1]')         # optional - magma
 0
```

**console()**

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

**EXAMPLES:**

```
sage: magma.console()            # not tested
Magma V2.14-9       Sat Oct 11 2008 06:36:41 on one       [Seed = 1157408761]
Type ? for help.  Type <Ctrl>-D to quit.
>  
Total time: 2.820 seconds, Total memory usage: 3.95MB
```
cputime(t=None)
Return the CPU time in seconds that has elapsed since this Magma session started. This is a floating point number, computed by Magma.

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

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

OUTPUT:
• float - seconds

EXAMPLES:

<table>
<thead>
<tr>
<th>sage: type(magma.cputime())</th>
<th># optional - magma</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;... 'float'&gt;</td>
<td></td>
</tr>
<tr>
<td>sage: magma.cputime()</td>
<td># random, optional - magma</td>
</tr>
<tr>
<td>1.9399999999999999</td>
<td></td>
</tr>
<tr>
<td>sage: t = magma.cputime()</td>
<td># optional - magma</td>
</tr>
<tr>
<td>sage: magma.cputime(t)</td>
<td># random, optional - magma</td>
</tr>
<tr>
<td>0.02</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>sage: magma.eval(&quot;a := 10;print 2+a;&quot;)</th>
<th># optional - magma</th>
</tr>
</thead>
<tbody>
<tr>
<td>'12'</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>sage: magma.eval(&quot;a := %s;%(10^10000))</th>
<th># optional - magma</th>
</tr>
</thead>
<tbody>
<tr>
<td>''</td>
<td></td>
</tr>
</tbody>
</table>

Verify that trac ticket #9705 is fixed:

<table>
<thead>
<tr>
<th>sage: nl=chr(10)</th>
<th># newline character</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage: magma.eval(</td>
<td># optional - magma</td>
</tr>
<tr>
<td>....: &quot;&lt;_x&gt;:=PolynomialRing(Rationals());&quot;+nl+</td>
<td></td>
</tr>
<tr>
<td>....: &quot;repeat&quot;+nl+</td>
<td></td>
</tr>
<tr>
<td>....: &quot;g:=3<em>b^2</em>x^4+18<em>c^2</em>x^3-6<em>b^2</em>x^2-6<em>b</em>c<em>x-b^3-9</em>c^2 where b:=Random([-10..10]) where c:=Random([-10..10]);&quot;+nl+</td>
<td></td>
</tr>
<tr>
<td>....: &quot;until g ne 0 and Roots(g) ne [];&quot;+nl+</td>
<td></td>
</tr>
<tr>
<td>....: &quot;print &quot;success&quot;;&quot;</td>
<td></td>
</tr>
<tr>
<td>'success'</td>
<td></td>
</tr>
</tbody>
</table>

Verify that trac ticket #11401 is fixed:

22.5. Other Examples 169
function_call(function, args=[], params={}, nvals=1)

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

INPUT:

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

OUTPUT: MagmaElement or tuple of nvals MagmaElement’s

EXAMPLES:

```python
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
```

Next, we illustrate multiple return values:

```python
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):
... Runtime error in :=: Expected to assign 3 value(s) but only computed 2 value(s)
```

get(var)

Get the value of the variable var.

INPUT:

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

OUTPUT:

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

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

get_nthreads()
Get the number of threads used in Magma.

EXAMPLES:

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

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.setVerbose("Groebner", 2) # optional - magma
sage: magma.getVerbose("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

ideal(L)
Return the Magma ideal defined by L.

INPUT:

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

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

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

load(filename)
Load the file with given filename using the ‘load’ command in the Magma shell.

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

INPUT:
  • filename - string

OUTPUT: output printed when loading the file

EXAMPLES:

sage: filename = os.path.join(SAGE_TMP, 'a.m')
sage: with open(filename, 'w') as f:
    ....: _ = f.write('function f(n) return n^2; end function;
                   \nprint "hi";')
sage: print(magma.load(filename))  # optional - magma
Loading ".../a.m"
hi
sage: magma('f(12)')  # optional - magma
144

objgens(value, gens)
Create a new object with given value and gens.

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

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

EXAMPLES:

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

Because of how Magma works you can use this to change the variable names of the generators of an object:
set\((var, value)\)
Set the variable var to the given value in the Magma interpreter.

INPUT:

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

EXAMPLES:

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

set\_nthreads\((n)\)
Set the number of threads used for parallelized algorithms in Magma.

INPUT:

\* n - number of threads

EXAMPLES:

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

set\_seed\((seed=None)\)
Set the seed for the Magma interpreter.

The seed should be an integer.

EXAMPLES:

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

set\_verbose\((type, level)\)
Set the verbosity level for a given algorithm, class, etc. in Magma.

INPUT:

\* type - string (e.g. ‘Groebner’)
\* level - integer \(\geq 0\)

EXAMPLES:
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:

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

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

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

AssignNames(names)

EXAMPLES:

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

assign_names(names)

EXAMPLES:

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

eval(*args)

Evaluate self at the inputs.

INPUT:

• *args – import arguments

OUTPUT: self(*args)

EXAMPLES:

```python
sage: f = magma('Factorization')  # optional - magma
sage: f.evaluate(15)  # optional - magma
[ [3, 1], [5, 1] ]
sage: f(15)  # optional - magma
[ [3, 1], [5, 1] ]
```
evaluate(*args)
Evaluate self at the inputs.

INPUT:
• *args – import arguments

OUTPUT: self(*args)

EXAMPLES:

```python
sage: f = magma('Factorization')  # optional - magma
sage: f.evaluate(15)  # optional - magma
[ <3, 1>, <5, 1> ]
sage: f(15)  # optional - magma
[ <3, 1>, <5, 1> ]
sage: f = magma('GCD')  # optional - magma
sage: f.evaluate(15,20)  # optional - magma
5
```

```python
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 ]
```

```python
g = magma(k).gen(1)  # optional -- magma
a
```

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

22.5. Other Examples 175
gen_names()
Return list of Magma variable names of the generators of self.

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

EXAMPLES:

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

gens()
Return generators for self.

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

AUTHORS:
• William Stein (2006-07-02)

EXAMPLES:

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

get_magma_attribute(attrname)
Return value of a given Magma attribute. This is like self[attrname] in Magma.

OUTPUT: MagmaElement

EXAMPLES:

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

Return the ideal of self with given list of generators.

INPUT:

• gens - object or list/tuple of generators

OUTPUT:

• magma element - a Magma ideal

EXAMPLES:

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

list_attributes()

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

OUTPUT: list of strings

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

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

methods(any=False)

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

INPUT:

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

OUTPUT: list of strings

EXAMPLES:

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

quo(gens, **args)

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

INPUT:

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

OUTPUT:
• **magma element** - the quotient object
• **magma element** - mapping from self to the quotient object

**EXAMPLES:**

```python
sage: V = magma('VectorSpace(RationalField(),3)')  # optional - magma
sage: V.quo([[1,2,3], [1,1,2]])  # optional - magma

(Full Vector space of degree 1 over Rational Field, Mapping from: Full Vector space of degree 3 over Rational Field to Full Vector space of degree 1 over Rational Field)
```

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

```python
sage: W = V.sub([[1,2,3], [1,1,2]])  # optional - magma
sage: V.quo(W)  # optional - magma

(Full Vector space of degree 1 over Rational Field, Mapping from: Full Vector space of degree 3 over Rational Field to Full Vector space of degree 1 over Rational Field)
```

We quotient a ZZ module out by a submodule.

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

**set_magma_attribute**(attrname, value)

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

**EXAMPLES:**

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

**sub**(gens)

**Return the sub-object of self with given gens.**

**INPUT:**
• gens - object or list/tuple of generators

**EXAMPLES:**

```python
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)

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

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

class sage.interfaces.magma.MagmaGBDefaultContext(magma=None)
    Bases: object
    Context to force preservation of verbosity options for Magma’s Groebner basis computation.

class sage.interfaces.magma.MagmaGBLogPrettyPrinter(verbosity=1, style='magma')
    Bases: object
    A device which filters Magma Groebner basis computation logs.

    flush()

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

    write(s)

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

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

    EXAMPLES:
    sage: sage.interfaces.magma.extcode_dir() 'dir_.../data/

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

    INPUT:
    - x - any object

    OUTPUT: bool

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

sage.interfaces.magma.magma_console()
Run a command line Magma session.

EXAMPLES:

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

sage.interfaces.magma.magma_gb_standard_options(func)
Decorator to force default options for Magma.

EXAMPLES:

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

sage.interfaces.magma.reduce_load_Magma()
Used in unpickling a Magma interface.

This functions just returns the global default Magma interface.

EXAMPLES:

```
sage: sage.interfaces.magma.reduce_load_Magma()
Magma
```
class sage.interfaces.magma_free.MagmaExpr
    Bases: str

class sage.interfaces.magma_free.MagmaFree
    Bases: object

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

    EXAMPLES:

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

eval(x, **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.

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

    EXAMPLES:

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

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

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

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

EXAMPLES:

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

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

## 24.1 Tutorial

AUTHORS:


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

   ```maple
   factor( (x^5-1));
   ```
We can write that in sage as

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

Notice, there is no need to use a semicolon.

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('factor(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('factor((x^12-1)/(x-1))').simplify() # optional - maple
x^11+x^10+x^9+x^8+x^7+x^6+x^5+x^4+x^3+x^2+x+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('factor((x^12-1))') # optional - maple
(x-1)*(x+1)*(x^2+x+1)*(x^2-x+1)*(x^2+1)*(x^4-x^2+1)
```

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

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

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

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

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

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

We see there are two choices. Type

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

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

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

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

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

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

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

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

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

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

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

```python
sage: maple.isprime(maple.fibonacci(27))  # optional - maple
false
sage: maple.ifactor(maple.fibonacci(27))  # optional - maple
```

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).

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

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

In Sage, we write

```python
sage: mysqcu = maple('proc(x) if x > 0 then x^2 else x^3 fi end')  # optional - maple
sage: mysqcu(5)  # optional - maple
25
sage: mysqcu(-5)  # optional - maple
-125
```
More complicated programs should be put in a separate file and loaded.

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

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

Interface to the Maple interpreter.

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

```python
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:

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

```python
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:

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

```python
console()
```

Spawn a new Maple command-line session.

EXAMPLES:

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

```python
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:

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

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

expect()

Return the pexpect object for this Maple session.

EXAMPLES:

sage: m = Maple()  # optional - maple
sage: m.expect() is None  # optional - maple
True
sage: m._start()  # optional - maple
sage: m.expect()  # optional - maple
<pexpect.spawn instance at 0x...>
sage: m.quit()  # optional - maple

get(var)

Get the value of the variable var.

EXAMPLES:

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

help(string)

Display Maple help about string.

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

INPUT:

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

EXAMPLES:

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

load(package)

Make a package of Maple procedures available in the interpreter.

INPUT:

- package - string

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

sage: maple.quit()  # reset maple; optional -- maple
sage: maple('partition(10)')  # optional - maple
partition(10)
sage: maple('bell(10)')  # optional - maple
bell(10)
set(var, value)
Set the variable var to the given value.

EXAMPLES:

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

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

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

INPUT:

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

EXAMPLES:

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

with_package(package)
Make a package of Maple procedures available in the interpreter.

INPUT:

- package - string

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

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

(continues on next page)
sage: maple('bell(10)')  # optional - maple
115975
sage: maple('fibonacci(10)')  # optional - maple
55

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

op(i)
Return the i-th operand of this expression.

INPUT:
• i – an integer

EXAMPLES:

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

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

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

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

EXAMPLES:

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

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

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

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

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

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

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

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

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

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

If you don’t want Sage to go to the trouble of creating a wrapper for the Mathematica expression, then you can call `mathematica.eval(expr)`, which returns the result as a Mathematica AsciiArtString formatted string. If you want the result to be a string formatted like Mathematica’s InputForm, call `repr(mobj)` on the wrapper object `mobj`. If you want a string formatted in Sage style, call `mobj._sage_repr()`:
Finally, if you just want to use a Mathematica command line from within Sage, the function mathematica_console() dumps you into an interactive command-line Mathematica session. This is an enhanced version of the usual Mathematica command-line, in that it provides readline editing and history (the usual one doesn’t!)

25.1 Tutorial

We follow some of the tutorial from http://library.wolfram.com/conferences/devconf99/withoff/Basic1.html/. For any of this to work you must buy and install the Mathematica program, and it must be available as the command math in your PATH.

25.1.1 Syntax

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

```python
sage: m = mathematica
sage: a = m(1) + m(1); a  # optional - mathematica
```

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

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

We can also compute 7(2 + 3).

```python
sage: m(7).Times(m(2).Plus(m(3)))  # optional - mathematica
sage: m('7(2+3)')  # optional - mathematica
```
25.1.2 Some typical input

We solve an equation and a system of two equations:

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

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

25.1.3 Assignments and definitions

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

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

The Sage interfaces changes Sage lists into Mathematica lists:

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

25.1.4 Function definitions

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

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

25.1. Tutorial
25.1.5 Numerical Calculations

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

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

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

```python
sage: gp('solve(x=1,2,exp(x)-3*x)')
1.512134551657842473896739678 # 32-bit
1.512134551678424738967396780720387046 # 64-bit
```

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

```python
sage: mathematica('FindMinimum[x^3 - 6x^2 + 11x - 5, \{x,3\}]') # optional - mathematica
{0.6150998205402516, \{x -> 2.577350269962973\}}
```

```python
sage: f = mathematica('x^3 - 6x^2 + 11x - 5') # optional - mathematica
sage: f.FindMinimum(['x', 3]) # optional - mathematica
{0.6150998205402516, \{x -> 2.577350269962973\}}
```

25.1.6 Polynomial and Integer Factorization

We factor a polynomial of degree 200 over the integers.

```python
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
```

```python
sage: g = mathematica(str(f)) # optional - mathematica
```

```python
sage: print(g) # optional - mathematica
2 100 101 200
100 + 315 x - 85 x^2 + 25 x^100 + 12 x^101 + x^200

sage: g # optional - mathematica
100 + 315*x - 85*x^2 + 25*x^100 + 12*x^101 + x^200
```

```python
sage: print(g.Factor()) # optional - mathematica
(20 - 5 x + x ) (5 + 17 x + x )
```

We can also factor a multivariate polynomial:

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

```python
sage: print(f.Factor()) # optional - mathematica
3 2 3
x (x - y) (-2 x + x + y )
```

We factor an integer:

```python
sage: n = mathematica(2434500) # optional - mathematica
```

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

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

Mathematica’s ECM package is no longer available.

## 25.2 Long Input

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

sage: t = '%s' % 10**10000  # ten thousand character string.
sage: a = mathematica(t)  # optional - mathematica
sage: a = mathematica.eval(t)  # optional - mathematica

## 25.3 Loading and saving

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

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

## 25.4 Complicated translations

The `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 `obj._sage_()` and supply a translation dictionary:

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

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

OTHER Examples:

```
    sage: def math_bessel_K(nu,x):
    ....:     return mathematica(nu).BesselK(x).N(20)
    sage: math_bessel_K(2,I)  # optional - mathematica
    -2.59288617549119697817 + 0.18048997206696202663*I
    sage: slist = [[1, 2], 3., 4 + I]
    sage: mlist = mathematica(slist); mlist  # optional - mathematica
    {{1, 2}, 3., 4 + I}
    sage: slist2 = list(mlist); slist2  # optional - mathematica
    [[1, 2], 3., 4 + I]
    sage: slist2[0]  # optional - mathematica
    {1, 2}
    sage: slist2[0].parent()  # optional - mathematica
    Mathematica
    sage: slist3 = mlist.sage(); slist3  # optional - mathematica
    [[1, 2], 3.000000000000000, I + 4]
    sage: mathematica('10.^80')  # optional - mathematica
    1.00000000000000e80
    sage: mathematica('10.^80').sage()  # optional - mathematica
    1.00000000000000e80
```

AUTHORS:

- William Stein (2005): first version
- Felix Lawrence (2009-08-21): Added support for importing Mathematica lists and floats with exponents.

```python
class sage.interfaces.mathematica.Mathematica(maxread=None, script_subdirectory=None, logfile=None, server=None, server_tmpdir=None, command=None, verbose_start=False):
    # Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.expect.Expect
    Interface to the Mathematica interpreter.
    chdir(dir)
    Change Mathematica’s current working directory.
    EXAMPLES:
```
sage: mathematica.chdir('/')  # optional - mathematica
sage: mathematica('Directory[]')  # optional - mathematica
"/

console(readline=True)

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

generate(var, ascii_art=False)
    Get the value of the variable var.

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

help(cmd)

set(var, value)
    Set the variable var to the given value.

class sage.interfaces.mathematica.MathematicaElement(parent, value, is_name=False, name=None)
    Bases: sage.interfaces.expect.ExpectElement

    n(*args, **kwargs)
        Numerical approximation by converting to Sage object first
        Convert the object into a Sage object and return its numerical approximation. See documentation of the function sage.misc.functional.n() for details.

        EXAMPLES:

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

    save_image(filename, ImageSize=600)
        Save a mathematica graphics

        INPUT:
            • filename – string. The filename to save as. The extension determines the image file format.
            • ImageSize – integer. The size of the resulting image.

        EXAMPLES:

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

25.4. Complicated translations
INPUT:
- ImageSize – integer. The size of the resulting image.

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

EXAMPLES:

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

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

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

```
sage: P.show(ImageSize=800) # optional - mathematica
```

```
˓→mathematicafrontend
```

```
˓→mathematicafrontend
```

```
str()
```

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

```
Bases: sage.interfaces.expect.ExpectFunction
```

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

```
Bases: sage.interfaces.expect.FunctionElement
```

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

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

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

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

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

```
OUTPUT:
list of unicode strings
```

```
EXAMPLES:
```
sage: from sage.interfaces.mathematica import request_wolfram_alpha
sage: from sage.interfaces.mathematica import parse_moutput_from_json
sage: page_data = request_wolfram_alpha('Integrate Sin[x]') # optional internet
sage: parse_moutput_from_json(page_data) # optional internet
[u'-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)
['-Cos[x]', '{x == 0}', '{x == Pi C[1], Element[C[1], Integers]}']
```

sage.interfaces.mathematica.reduce_load(X)
sage.interfaces.mathematica.request_wolfram_alpha(input, verbose=False)

Request Wolfram Alpha website.

INPUT:

• input – string
• verbose – bool (default: False)

OUTPUT:

json

EXAMPLES:

```python
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',
 'encryptedEvaluatExpression',
 'encryptedParsedExpression',
 'error',
 'host',
 'id',
 'inputstring',
 'numpods',
 'parsetimedout',
 'parsedtimings',
 'pods',
 'recalculate',
 'related',
 'server',
 'sponsorCategories',
 'success',
 'timedout',
 'timedoutpods',
 'timing',
 'version']
```

sage.interfaces.mathematica.symbolic_expression_from_mathematica_string(mexpr)

Translate a mathematica string into a symbolic expression

INPUT:

• mexpr – string

OUTPUT:

symbolic expression

EXAMPLES:

```python
sage: from sage.interfaces.mathematica import symbolic_expression_from_mathematica_string
sage: symbolic_expression_from_mathematica_string(u'-Cos[x]')
-cos(x)
```
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)`.

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

```
sage: mobj.Factor()
(-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
sage: mobj = mathics(x^2-1)
```

```sage
sage: mobj2 = mobj.Factor(); mobj2
(-1 + x) (1 + x)
```

```sage
Mathics
```

```sage
sage: mobj2.parent()
# optional - mathics
```

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

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

### 26.1 Tutorial

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

#### 26.1.1 Syntax

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

The following is equivalent to \texttt{Plus[2, 3]} in Mathics:

We can also compute \texttt{7(2+3)}.
26.1.2 Some typical input

We solve an equation and a system of two equations:

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

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

26.1.3 Assignments and definitions

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

```sage
c = m(5) # optional - mathics
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
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}}
```

26.1.4 Function definitions

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

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

81
26.1.5 Numerical Calculations

We find the \( x \) such that \( e^x - 3x = 0 \).

```
sage: eqn = mathics('Exp[x] - 3x == 0') # optional - mathics
sage: eqn.FindRoot([x', 2]) # optional - mathics
{x -> 1.51213}
```

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

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

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

```
sage: mathics('FindMinimum[x^3 - 6x^2 + 11x - 5, {x,3}]') # not tested (since not
˓→supported, so far)
{0.6150998205402516, {x -> 2.5773502699629733}}
sage: f = mathics('x^3 - 6x^2 + 11x - 5') # optional - mathics
sage: f.FindMinimum([x', 3]) # not tested (since not
˓→supported, so far)
{0.6150998205402516, {x -> 2.5773502699629733}}
```

26.1.6 Polynomial and Integer Factorization

We factor a polynomial of degree 200 over the integers.

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

We can also factor a multivariate polynomial:

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

We factor an integer:

```
sage: n = mathics(2434500) # optional - mathics
sage: n.FactorInteger() # optional - mathics
{{2, 2}, {3, 2}, {5, 3}, {541, 1}}
sage: F = n.FactorInteger(); F # optional - mathics
(continues on next page)
26.2 Long Input

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

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

26.3 Loading and saving

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

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

26.4 Complicated translations

The `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 \texttt{mobj\_\_sage\_()} and supply a translation dictionary:

\begin{verbatim}
sage: x = var('x')
sage: m = mathics('NewFn[x]')  # optional - mathics
sage: m\_\_sage\_({'NewFn': sin, 'x':x})  # optional - mathics
sin(x)
\end{verbatim}

For more details, see the documentation for \texttt{\_\_sage\_()}.

OTHER Examples:

\begin{verbatim}
sage: def math_bessel_K(nu,x):
....:     return mathics(nu).BesselK(x).N(20)
sage: math_bessel_K(2,I)  # optional - mathics
-2.5928861754911969782 + 0.18048997206696202663 I
sage: slist = [[1, 2], 3., 4 + I]
sage: mlist = mathics(slist); mlist  # optional - mathics
{{1, 2}, 3., 4 + I}
sage: slist2 = list(mlist); slist2  # optional - mathics
[[1, 2], 3., 4 + I]
sage: slist2[0]  # optional - mathics
{1, 2}
sage: slist2[0].parent()  # optional - mathics
Mathics
sage: slist3 = mlist.sage(); slist3  # optional - mathics
[[1, 2], 3.00000000000000, 4.00000000000000 + 1.00000000000000*I]
sage: mathics('10.^80')  # optional - mathics
1.00000000000000e80
sage: mathics('10.^80').sage()  # optional - mathics
1.00000000000000e80
\end{verbatim}

AUTHORS:

- Sebastian Oehms (2021): first version from a copy of the Mathematica interface (see trac ticket \#31778).

Thanks to Rocky Bernstein and Juan Mauricio Matera for their support. For further acknowledgments see this list.

class \texttt{sage\_interfaces\_mathics.Mathics}(maxread=None, logfile=None, init_list_length=1024, seed=None)

Bases: \texttt{sage\_interfaces\_interface.Interface}

Interface to the Mathics interpreter.

Implemented according to the Mathematica interface but avoiding Pexpect functionality.

EXAMPLES:

\begin{verbatim}
sage: t = mathics('Tan[I + 0.5]')  # optional - mathics
Mathics
sage: t.parent()  # optional - mathics
sage: ts = t.sage()  # optional - mathics
sage: ts.parent()  # optional - mathics
Complex Field with 53 bits of precision
\end{verbatim}
More examples can be found in the module header.

**chdir(dir)**
Change Mathics's current working directory.

```python
sage: mathics.chdir('/')  # optional - mathics
```

**console()**
Spawn a new Mathics command-line session.

```python
sage: mathics.console()  # not tested
```

**eval(code, *args, **kwds)**
Evaluates a command inside the Mathics interpreter and returns the output in printable form.

```python
sage: mathics.eval('1+1')  # optional - mathics
'2'
```
get(var)
Get the value of the variable var.

EXAMPLES:

```python
sage: mathics.set('u', '2*x +E')  # optional - mathics
sage: mathics.get('u')            # optional - mathics
'E + 2 x'
```

help(cmd, long=False)
Return the Mathics documentation of the given command.

EXAMPLES:

```python
sage: mathics.help('Sin')         # optional - mathics
"\n 'Sin[z]\n returns the sine of z."

sage: print(_)                    # optional - mathics
'Sin[z]'
 returns the sine of z.

sage: print(mathics.help('Sin', long=True)) # optional - mathics
'Sin[z]'
 returns the sine of z.

Attributes[Sin] = {Listable, NumericFunction, Protected}

sage: print(mathics.Factorial.__doc__) # optional - mathics
'Factorial[n]'
 'n!'
 computes the factorial of n.

sage: u = mathics('Pi')            # optional - mathics
sage: print(u.Cos.__doc__)         # optional - mathics
'Cos[z]'
 returns the cosine of z.
```

set(var, value)
Set the variable var to the given value.

EXAMPLES:

```python
sage: mathics.set('u', '2*x +E')  # optional - mathics
sage: bool(mathics('u').sage() == 2*x+e) # optional - mathics
True
```

class sage.interfaces.mathics.MathicsElement(parent, value, is_name=False, name=None)
Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.interface.InterfaceElement
Element class of the Mathics interface.

Its instances are usually constructed via the instance call of its parent. It wraps the Mathics library for this object. In a session Mathics methods can be obtained using tab completion.

EXAMPES:

```
sage: me=mathics(e); me               # optional - mathics
E
sage: type(me)                        # optional - mathics
<class 'sage.interfaces.mathics.MathicsElement'>
sage: P = me.parent(); P               # optional - mathics
Mathics
sage: type(P)                         # optional - mathics
<class 'sage.interfaces.mathics.Mathics'>
```

Access to the Mathics expression objects:

```
sage: res = me._mathics_result         # optional - mathics
sage: type(res)                        # optional - mathics
<class 'mathics.core.evaluation.Result'>
sage: expr = res.last_eval; expr      # optional - mathics
<Symbol: SystemE>
sage: type(expr)                      # optional - mathics
<class 'mathics.core.expression.Symbol'>
```

Applying Mathics methods:

```
sage: me.to_sympy()                    # optional - mathics
E
sage: me.get_name()                   # optional - mathics
'SystemE'
sage: me.is_inexact()                # optional - mathics
False
sage: me.is_symbol()                  # optional - mathics
True
```

Conversion to Sage:

```
sage: bool(me.sage() == e)            # optional - mathics
True
```

**n(***args, **kwargs**)

Numerical approximation by converting to Sage object first

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

EXAMPLES:

```
sage: mathics('Pi').n(10)              # optional -- mathics
3.1
sage: mathics('Pi').n()               # optional -- mathics
3.14159265358979
sage: mathics('Pi').n(digits=10)      # optional -- mathics
3.141592654
```
save_image(filename, ImageSize=600)
Save a mathics graphics

INPUT:

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

EXAMPLES:

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

show(ImageSize=600)
Show a mathics expression immediately.

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

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

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

sage.interfaces.mathics.mathics_console()
Spawn a new Mathics command-line session.

EXAMPLES:

```
sage: mathics_console()  # not tested
```

Mathics 2.1.1.dev0
on CPython 3.9.2 (default, Mar 19 2021, 22:23:28)
using SymPy 1.7, mpmath 1.2.1, numpy 1.19.5, cython 0.29.21

Copyright (C) 2011-2021 The Mathics Team.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions.
See the documentation for the full license.
Quit by evaluating Quit[] or by pressing CONTROL-D.

(continues on next page)
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
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 =

 4
```

```
sage: a = matlab(10)  # optional - matlab
sage: a**10  # optional - matlab
1.0000e+10
```

**AUTHORS:**
- William Stein (2006-10-11)

### 27.1 Tutorial

**EXAMPLES:**

```
sage: matlab('4+10')  # optional - matlab
14
```

```
sage: matlab('date')  # optional - matlab; random output
18-Oct-2006
```

```
sage: matlab('5*10 + 6')  # optional - matlab
56
```

```
sage: matlab('(6+6)/3')  # optional - matlab
4
```

```
sage: matlab('9')**2  # optional - matlab
81
```

```
sage: a = matlab(10); b = matlab(20); c = matlab(30)  # optional - matlab
sage: avg = (a+b+c)/3 ; avg  # optional - matlab
20
```

```
sage: parent(avg)  # optional - matlab
Matlab
```

sage: my_scalar = matlab('3.1415')  # optional - matlab
sage: my_scalar
3.1415
sage: my_vector1 = matlab('[1,5,7]')  # optional - matlab
sage: my_vector1
1 5 7
sage: my_vector2 = matlab('[1;5;7]')  # optional - matlab
sage: my_vector2
1 5
7
sage: my_vector1 * my_vector2  # optional - matlab
75
sage: row_vector1 = matlab('[1 2 3]')  # optional - matlab
sage: row_vector2 = matlab('[3 2 1]')  # optional - matlab
sage: matrix_from_row_vec = matlab('[%s; %s]'%(row_vector1.name(), row_vector2.name())) # optional - matlab
sage: matrix_from_row_vec
1 2 3
3 2 1
sage: column_vector1 = matlab('[1;3]')  # optional - matlab
sage: column_vector2 = matlab('[2;8]')  # optional - matlab
sage: matrix_from_col_vec = matlab('[%s %s]'%(column_vector1.name(), column_vector2.name()))  # optional - matlab
sage: matrix_from_col_vec
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
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
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
0.5000 2.5000 4.5000 6.5000 8.5000
Matrix indexing works as follows:

```sage
g = my_vector1(1)  # optional - matlab
1

g = my_vector1(2)  # optional - matlab
5

g = my_vector1(3)  # optional - matlab
7
```

Setting using parenthesis cannot work (because of how the Python language works). Use square brackets or the set function:

```sage
g = my_matrix(3, 2)  # optional - matlab
4
```

class sage.interfaces.matlab.Matlab(maxread=None, script_subdirectory=None, logfile=None, server=None, server_tmpdir=None):

Bases: sage.interfaces.expect.Expect

Interface to the Matlab interpreter.

EXAMPLES:

```sage
g = a * b

30
122
505
```

chdir(directory)

Change MATLAB’s current working directory.

EXAMPLES:

```sage
g = matlab.chdir('/')  # optional - matlab
/
```

c = get(var)

Get the value of the variable var.
EXAMPLES:

```python
sage: s = matlab.eval('a = 2')  # optional - matlab
sage: matlab.get('a')          # optional - matlab
'2'
```

`sage2matlab_matrix_string(A)`

Return an matlab matrix from a Sage matrix.

INPUT: A Sage matrix with entries in the rationals or reals.

OUTPUT: A string that evaluates to an Matlab matrix.

EXAMPLES:

```python
sage: M33 = MatrixSpace(QQ,3,3)
sage: A = M33([1,2,3,4,5,6,7,8,0])
sage: matlab.sage2matlab_matrix_string(A)  # optional - matlab
'[1, 2, 3; 4, 5, 6; 7, 8, 0]'
```

AUTHOR:

• David Joyner and William Stein

`set(var, value)`

Set the variable var to the given value.

`strip_answer(s)`

Returns the string s with Matlab’s answer prompt removed.

EXAMPLES:

```python
sage: s = '\nans =\n
2\n'
sage: matlab.strip_answer(s)
'2'
```

`version()`

`whos()`

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

Bases: `sage.interfaces.expect.ExpectElement`

`set(i, j, x)`

`sage.interfaces.matlab.matlab_console()`

This requires that the optional matlab program be installed and in your PATH, but no optional Sage packages need be installed.

EXAMPLES:

```python
sage: matlab_console()  # optional - matlab; not tested
< M A T L A B >
Copyright 1984-2006 The MathWorks, Inc.
...
>> 2+3

ans =
5
quit
```
Typing quit exits the matlab console and returns you to Sage. matlab, like Sage, remembers its history from one session to another.

```
sage.interfaces.matlab.matlab_version()
Return the version of Matlab installed.
```

```
sage: matlab_version()  # random; optional - matlab
'7.2.0.283 (R2006a)'
```

```
sage.interfaces.matlab.reduce_load_Matlab()
```
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

This is the interface used by the maxima object:

```python
sage: type(maxima)
<class 'sage.interfaces.maxima.Maxima'>
```

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

EXAMPLES: We evaluate a very simple expression in Maxima.

```python
sage: maxima('3 * 5')
15
```

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

```python
sage: F = maxima.factor('x^5 - y^5')
sage: F
-(y-x)*(y^4+x*y^3+x^2*y^2+x^3*y+x^4)
sage: type(F)
<class 'sage.interfaces.maxima.MaximaElement'>
```

Note that Maxima objects can also be displayed using “ASCII art”; to see a normal linear representation of any Maxima object x. Just use the print command: use `str(x)`.
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.

You 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.

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.

28.1 Tutorial

We follow the tutorial at [http://maxima.sourceforge.net/docs/intromax/intromax.html](http://maxima.sourceforge.net/docs/intromax/intromax.html).
sage: maxima.eval('fpprec : 100')
100
sage: a.bfloat()
8.
˓→201219330881975641524897300208124278520484385931494122123712401731214875401104126661238...

sage: maxima('100!')
9332621544394415268169923885626670049071596826438162146859263895217599932299156089414639...

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()
(3*y*z^2+5*z+25*y)^3/z^6

Here is an example of solving an algebraic equation:

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]]

You can even nicely typeset the solution in latex:

sage: latex(s)
\left[ \left[ a=-\frac{\sqrt{79}\,i-11}{4} , b=\frac{\sqrt{79}\,i+9 }{4} , c=\frac{\sqrt{79}\,i+1}{10} \right] , \left[ a=\frac{\sqrt{79}\,i+11}{4} , b=-\frac{\sqrt{79}\,i-9}{4} , c=-\frac{\sqrt{79}\,i-1}{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

28.1. Tutorial
\[ f(x) = x^3 \cdot e^{kx} \cdot \sin(wx) \]
\[ f'(x) = kx^3 \cdot e^{kx} \cdot \sin(wx) + 3x^2 \cdot e^{kx} \cdot \sin(wx) + w^2 \cdot x^3 \cdot e^{kx} \cdot \cos(wx) \]
\[ f(x) = \frac{1}{x^2} \]
\[ f'(x) = \frac{1}{x} \]
\[ f = \frac{f}{\sinh(kx)^4} \]
\[ f'(x) = \frac{f}{\sinh(kx)} \cdot \frac{1}{x^2} \]
\[ f(x) = \frac{f}{\sinh(kx)^4} \cdot \frac{1}{(2x^2) + (11x^2)} + \frac{45 - (62x^2f) \cdot x^2}{945} \]

**28.2 Examples involving matrices**

We illustrate computing with the matrix whose \( i, j \) entry is \( \frac{i}{j} \), for \( i, j = 1, \ldots, 4 \).

\[ A = \begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 \\ 2 & 1 & 2/3 & 1/2 \\ 3 & 3/2 & 1 & 3/4 \\ 4 & 2 & 4/3 & 1 \end{bmatrix} \]

\[ A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \]

\[ \text{eigenvectors} = \begin{bmatrix} 1,0,0,-4 \\ 0,1,0,-2 \\ 0,0,1,-4/3 \\ 1,2,3,4 \end{bmatrix} \]

We can also compute the echelon form in Sage:

\[ B = \begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \]

\[ B = \begin{bmatrix} 1,1/2,1/3,1/4 \\ 0,0,0,0 \\ 0,0,0,0 \end{bmatrix} \]

\[ B = \begin{bmatrix} 0,4 \\ 3,1 \end{bmatrix} \]

\[ B = \begin{bmatrix} [0,4],[3,1] \end{bmatrix} \]
28.3 Laplace Transforms

We illustrate Laplace transforms:

```python
sage: _ = maxima.eval("f(t) := t*sin(t)"
sage: maxima("laplace(f(t),t,s)"
(2*s)/(s^2+1)^2

sage: maxima("laplace(delta(t-3),t,s)"
#Dirac delta function
%e^-(3*s)

sage: _ = maxima.eval("f(t) := exp(t)*sin(t)"
sage: maxima("laplace(f(t),t,s)"
1/(s^2-2*s+2)

sage: _ = maxima.eval("f(t) := t^5*exp(t)*sin(t)"
sage: maxima("laplace(f(t),t,s)"
(360*(2*s-2))/(s^2-2*s+2)^4-(480*(2*s-2)^3)/(s^2-2*s+2)^5 +(120*(2*s-2)^5)/(s^2-2*s+2)^6

sage: print(maxima("laplace(diff(x(t),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,2),t,s)"
3*s^2*laplace(x(t),t,s)-x(0)

sage: maxima("laplace(diff(x(t),t,2),t,s)"
(~%at('diff(x(t),t,1),t = 0))+s^2*laplace(x(t),t,s)-x(0)*s

It is difficult to read some of these without the 2d representation:

```
28.4 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)"")
[9,1,2,1,2,1,17,1,2,1,2,1,17,1,2,1,2,1,17,1,2,1,2,1,17,2]
```

28.5 Special examples

In this section we illustrate calculations that would be awkward to do (as far as I know) in non-symbolic computer algebra systems like MAGMA or GAP.

We compute the \( \gcd \) of \( 2x^{n+4} - x^{n+2} \) and \( 4x^{n+1} + 3x^n \) for arbitrary \( n \).

```
sage: f = maxima('2*x^(n+4) - x^(n+2)')
sage: g = maxima('4*x^(n+1) + 3*x^n')
sage: f.gcd(g)
x^n
```

You can plot 3d graphs (via gnuplot):

```
sage: maxima('plot3d(x^2-y^2, [x,-2,2], [y,-2,2], [grid,12,12])')  # not tested
[displays a 3 dimensional graph]
```

You can formally evaluate sums (note the \texttt{nusum} command):

```
sage: S = maxima('nusum(exp(1+2*i/n),i,1,n)')
sage: print(S)
    2/n + 3
%e ---------------------------------------------
    2/n + 1
%e

    1/n  1/n
(%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
```

224 Chapter 28. Pexpect interface to Maxima
28.6 Miscellaneous

Obtaining digits of \( \pi \):

```sage
maxima.eval('fpprec : 100')
```

```sage
maxima(pi).bfloat()
```

```
3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348
```

Defining functions in maxima:

```sage
maxima.eval('fun[a] := a^2')
```

```sage
maxima('fun[10]')
```

28.7 Interactivity

Unfortunately maxima doesn’t seem to have a non-interactive mode, which is needed for the Sage interface. If any Sage call leads to maxima interactively answering questions, then the questions can’t be answered and the maxima session may hang. See the discussion at http://www.ma.utexas.edu/pipermail/maxima/2005/011061.html for some ideas about how to fix this problem. An example that illustrates this problem is `maxima.eval('integrate (exp(a*x), x, 0, inf)')`.

28.8 Latex Output

To TeX a maxima object do this:

```sage
latex(maxima('sin(u) + sinh(v^2)'))
```

```
\sinh v^2+\sin u
```

Here's another example:

```sage
g = maxima('exp(3*i*x)/(6*i) + exp(%i*x)/(2*i) + c')
```

```sage
latex(g)
```

```
-{{i,e^{3\,i\,x}}\over{6}}-{{i\,e^{i\,x}}\over{2}}+c
```

28.9 Long Input

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

Note: Using `maxima.eval` for long input is much less robust, and is not recommended.

```sage
t = "%s" %10^10000 # ten thousand character string.
```

```sage
a = maxima(t)
```
class \texttt{sage.interfaces.maxima.Maxima}(script\_subdirectory=None, logfile=None, server=None, init\_code=None)

Bases: \texttt{sage.interfaces.maxima\_abstract.MaximaAbstract}, \texttt{sage.interfaces.expect.Expect}

Interface to the Maxima interpreter.

\textbf{EXAMPLES:}

\begin{verbatim}
\sage: m = Maxima()
\sage: m == maxima
False
\end{verbatim}

\textbf{clear}(\texttt{var})

Clear the variable named \texttt{var}.

\textbf{EXAMPLES:}

\begin{verbatim}
\sage: maxima.set('xxxxx', '2')
\sage: maxima.get('xxxxx')
'2'
\sage: maxima.clear('xxxxx')
\sage: maxima.get('xxxxx')
'xxxxx'
\end{verbatim}

\textbf{get}(\texttt{var})

Get the string value of the variable \texttt{var}.

\textbf{EXAMPLES:}

\begin{verbatim}
\sage: maxima.set('xxxxx', '2')
\sage: maxima.get('xxxxx')
'2'
\end{verbatim}

\textbf{lisp}(\texttt{cmd})

Send a lisp command to Maxima.

\textbf{Note:} The output of this command is very raw - not pretty.

\textbf{EXAMPLES:}

\begin{verbatim}
\sage: maxima.lisp("(+ 2 17)") # random formatted output
:lisp (+ 2 17)
19
\end{verbatim}

\textbf{set}(\texttt{var}, \texttt{value})

Set the variable \texttt{var} to the given value.

\textbf{INPUT:}

\begin{itemize}
\item \texttt{var} - string
\item \texttt{value} - string
\end{itemize}

\textbf{EXAMPLES:}
```
sage: maxima.set('xxxx', '2')
sage: maxima.get('xxxx')
'2'
```

**set_seed** *(seed=None)*

make_random_state (n) returns a new random state object created from an integer seed value equal to n modulo 2^32. n may be negative.  

EXAMPLES:

```
sage: m = Maxima()
sage: m.set_seed(1)
1
sage: [m.random(100) for i in range(5)]
[45, 39, 24, 68, 63]
```

**class**  
sage.interfaces.maxima.MaximaElement *(parent, value, is_name=False, name=None)*


Element of Maxima through Pexpect interface.  

EXAMPLES:  

Elements of this class should not be created directly. The targeted parent should be used instead:

```
sage: maxima(3)
3
sage: maxima(cos(x)+e^234)
cos(_SAGE_VAR_x)+%e^234
```

**display2d** *(onscreen=True)*

Return the 2d string representation of this Maxima object.  

EXAMPLES:

```
sage: F = maxima('x^5 - y^5').factor()
sage: F.display2d()

\[ 4 3 2 2 3 4 \]
\[ -(y - x)(y + x y + x y + x y + x) \]
```

**class**  
sage.interfaces.maxima.MaximaElementFunction *(parent, name, defn, args, latex)*

Bases: sage.interfaces.maxima_abstract.MaximaAbstractElementFunction  

Maxima user-defined functions.  

EXAMPLES:  

Elements of this class should not be created directly. The method function of the targeted parent should be used instead:

```
sage: maxima.function('x,y','h(x)*y')

h(x)*y
```

**sage.interfaces.maxima.is_MaximaElement** *(x)*

Returns True if x is of type MaximaElement.  

EXAMPLES:  

28.9. Long Input
```python
sage: from sage.interfaces.maxima import is_MaximaElement
sage: m = maxima(1)
True
sage: is_MaximaElement(1)
False
```

`sage.interfaces.maxima.reduce_load_Maxima()`  
Unpickle a Maxima Pexpect interface.

**EXAMPLES:**

```python
sage: from sage.interfaces.maxima import reduce_load_Maxima
sage: reduce_load_Maxima()
Maxima
```

`sage.interfaces.maxima.reduce_load_Maxima_function(parent, defn, args, latex)`  
Unpickle a Maxima function.

**EXAMPLES:**

```python
sage: from sage.interfaces.maxima import reduce_load_Maxima_function
sage: f = maxima.function('x,y','sin(x+y)')
sage: _,args = f.__reduce__()
```

```python
sage: g = reduce_load_Maxima_function(*args)
```

```python
sage: g == f
True
```
CHAPTER TWENTYNINE

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

This is an abstract class implementing the functions shared between the Pexpect and library interfaces to Maxima.

class sage.interfaces.maxima_abstract.MaximaAbstract(name='maxima_abstract')

Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.interface.Interface

Abstract interface to Maxima.

INPUT:

• name - string

OUTPUT: the interface

EXAMPLES:

This class should not be instantiated directly, but through its subclasses Maxima (Pexpect interface) or MaximaLib (library interface):

```python
sage: m = Maxima()
sage: from sage.interfaces.maxima_abstract import MaximaAbstract
sage: isinstance(m, MaximaAbstract)
True
```

```python
chdir(dir)
```

Change Maxima's current working directory.

INPUT:
• dir - string
OUTPUT: none
EXAMPLES:
\begin{verbatim}
sage: maxima.chdir('/')
\end{verbatim}

\textbf{completions}(s, \textit{verbose}=True)
Return all commands that complete the command starting with the string \textit{s}. This is like typing \textit{s}[tab] in the Maxima interpreter.

INPUT:
• \textit{s} - string
• \textit{verbose} - boolean (default: True)

OUTPUT: array of strings
EXAMPLES:
\begin{verbatim}
sage: sorted(maxima.completions('gc', verbose=False)) ['gcd', 'gcdex', 'gcfactor', 'gctime']
\end{verbatim}

\textbf{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 \texttt{maxima.interact()}.

INPUT: none
OUTPUT: none
EXAMPLES:
\begin{verbatim}
sage: maxima.console() # not tested (since we can't)
Maxima 5.34.1 http://maxima.sourceforge.net
Using Lisp ECL 13.5.1
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
This is a development version of Maxima. The function bug_report() provides bug reporting information.
(%i1)
sage: maxima.interact() # this is not tested either
   --> Switching to Maxima <--
maxima: 2+2
4
maxima:
   --> Exiting back to Sage <--
\end{verbatim}

\textbf{cputime}(t=None)
Returns the amount of CPU time that this Maxima session has used.

INPUT:
• \textit{t} - float (default: None): If \texttt{var(t)} is not None, then it returns the difference between the current CPU time and \texttt{var(t)}.

OUTPUT: float
EXAMPLES:

```
sage: t = maxima.cputime()
sage: _ = maxima.de_solve('diff(y,x,2) + 3*x = y', ['x', 'y'], [1,1,1])
sage: maxima.cputime(t) # output random
0.568913
```

`de_solve(de, vars, ics=None)`
Solves a 1st or 2nd order ordinary differential equation (ODE) in two variables, possibly with initial conditions.

**INPUT:**
- `de` - a string representing the ODE
- `vars` - a list of strings representing the two variables.
- `ics` - a triple of numbers `[a,b1,b2]` representing `y(a)=b1, y'(a)=b2`

**EXAMPLES:**

```
sage: maxima.de_solve('diff(y,x,2) + 3*x = y', ['x', 'y'], [1,1,1])
y = 3*x-2*%e^(x-1)
sage: maxima.de_solve('diff(y,x,2) + 3*x = y', ['x', 'y'])
y = %k1*%e^x+%k2*%e^-x+3*x
sage: maxima.de_solve('diff(y,x) + 3*x = y', ['x', 'y'])
y = (%c-3*((-x)-1)*%e^-x)*%e^x
sage: maxima.de_solve('diff(y,x) + 3*x = y', ['x', 'y'],[1,1])
y = -%e^-1*(5*%e^x-3*%e*x-3*%e)
```

`de_solve_laplace(de, vars, ics=None)`
Solves an ordinary differential equation (ODE) using Laplace transforms.

**INPUT:**
- `de` - a string representing the ODE (e.g., `de = "diff(f(x),x,2)=diff(f(x),x)+sin(x)"`)  
- `vars` - a list of strings representing the variables (e.g., `vars = ["x","f"]`)  
- `ics` - a list of numbers representing initial conditions, with symbols allowed which are represented by strings (eg, `f(0)=1, f'(0)=2` is `ics = [0,1,2]`)  

**EXAMPLES:**

```
sage: maxima.clear('x'); maxima.clear('f')
sage: maxima.de_solve_laplace("diff(f(x),x,2) = 2*diff(f(x),x)-f(x)", ["x","f"], [0,1,2])
f(x) = x*%e^x+%e^x
sage: maxima.clear('x'); maxima.clear('f')
sage: f = maxima.de_solve_laplace("diff(f(x),x,2) = 2*diff(f(x),x)-f(x)", ["x","f"])
sage: f
f(x) = x*%e^x*(at('diff(f(x),x,1),x = 0)) - f(0)*x*%e^x + f(0)*%e^x
sage: print(f)
 | x d x d   x d
f(x) = x %e (--- (f(x))! ) - f(0) x %e + f(0) %e
```

(continues on next page)
\[ \frac{dx}{dt} !x = 0 \]

**Note:** The second equation sets the values of \( f(0) \) and \( f'(0) \) in Maxima, so subsequent ODEs involving these variables will have these initial conditions automatically imposed.

demo\( (s) \)
Run Maxima's demo for \( s \).

**INPUT:**

\* \( s \) - string

**OUTPUT:** none

**EXAMPLES:**

```sage```
maxima.demo('cf') # not tested
read and interpret file: .../share/maxima/5.34.1/demo/cf.dem
```
At the '_' prompt, type ';' and <enter> to get next demonstration.
frac1:cf([1,2,3,4])
...
```
describe\( (s) \)
Return Maxima's help for \( s \).

**INPUT:**

\* \( s \) - string

**OUTPUT:**

Maxima's help for \( s \)

**EXAMPLES:**

```sage```
maxima.help('gcd')
-- Function: gcd (<p_1>, <p_2>, <x_1>, ...)
```
example\( (s) \)
Return Maxima's examples for \( s \).

**INPUT:**

\* \( s \) - string

**OUTPUT:**

Maxima's examples for \( s \)

**EXAMPLES:**

```sage```
maxima.example('arrays')
a[n]:=n*a[n-1]
```
(continues on next page)
function\((args, defn, rep=None, latex=None)\)

Return the Maxima function with given arguments and definition.

INPUT:

- **args** - a string with variable names separated by commas
- **defn** - a string (or Maxima expression) that defines a function of the arguments in Maxima.
- **rep** - an optional string; if given, this is how the function will print.

OUTPUT: Maxima function

EXAMPLES:

```sage
sage: f = maxima.function('x', 'sin(x)')
sage: f(3.2)  # abs tol 2e-16
-0.058374143427579909
sage: f = maxima.function('x,y', 'sin(x)+cos(y)')
sage: f(2, 3.5)  # abs tol 2e-16
sin(2)-0.9364566872907963
sage: f
sin(x)+cos(y)
sage: g = f.integrate('z')
sage: g
(cos(y)+sin(x))*z
sage: g(1,2,3)
3*(cos(2)+sin(1))
```

The function definition can be a Maxima object:

```sage
sage: an_expr = maxima('sin(x)*gamma(x)')
sage: t = maxima.function('x', an_expr)
sage: t
gamma(x)*sin(x)
sage: t(2)
sin(2)
sage: float(t(2))
0.9092974268256817
sage: loads(t.dumps())
gamma(x)*sin(x)
```

help\((s)\)

Return Maxima’s help for s.
INPUT:

• s - string

OUTPUT:

Maxima’s help for s

EXAMPLES:

```
sage: maxima.help('gcd')
-- Function: gcd (<p_1>, <p_2>, <x_1>, ...)
...
```

```
plot2d(*args)
Plot a 2d graph using Maxima / gnuplot.
maxima.plot2d(f, ['var', 'min', 'max'], options)
```

INPUT:

• f - a string representing a function (such as f="sin(x)"") [var, xmin, xmax]

• options - an optional string representing plot2d options in gnuplot format

EXAMPLES:

```
sage: maxima.plot2d('sin(x)', '[x,-5,5]')  # not tested
sage: opts = ['gnuplot_preamble, "set nokey"', 'gnuplot_term, ps', 'gnuplot_out_file, "sin-plot.eps"]
sage: maxima.plot2d('sin(x)', '[x,-5,5]', opts)  # not tested
```

The eps file is saved in the current directory.

```
plot2d_parametric(r, var, trange, nticks=50, options=None)
Plot r = [x(t), y(t)] for t = tmin...tmax using gnuplot with options.
```

INPUT:

• r - a string representing a function (such as r="[x(t),y(t)]")

• var - a string representing the variable (such as var = “t”)

• trange - [tmin, tmax] are numbers with tmintmax

• nticks - int (default: 50)

• options - an optional string representing plot2d options in gnuplot format

EXAMPLES:

```
sage: maxima.plot2d_parametric(['sin(t)'],['cos(t)'], "t", [-3.1,3.1])  # not tested
sage: opts = ['gnuplot_preamble, "set nokey"', 'gnuplot_term, ps', 'gnuplot_out_file, "circle-plot.eps"]
sage: maxima.plot2d_parametric(['sin(t)'],['cos(t)'], "t", [-3.1,3.1], opts)  # not tested
```

The eps file is saved to the current working directory.

Here is another fun plot:
plot3d(*args*)
Plot a 3d graph using Maxima / gnuplot.

maxima.plot3d(f, ['x, xmin, xmax'], ['y, ymin, ymax'], ['grid, nx, ny'], options)

INPUT:
- \( f \) - a string representing a function (such as \( f=\sin(x) \)) \([\text{var, min, max}]\)
- \( \text{args} \) should be of the form \([x, \text{xmin, xmax}], [y, \text{ymin, ymax}], [\text{grid, nx, ny}], \text{options} \)

EXAMPLES:

```sage`
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-cos-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 \( u = \text{umin...umax}, v = \text{vmin...vmax} \) using gnuplot with options.

INPUT:
- \( x, y, z \) - a string representing a function (such as \( x=\text{u}^2+\text{v}^2 \), ...) \( \text{vars} \) is a list or two strings representing variables (such as \( \text{vars} = [\text{"u"}, \text{"v"}] \))
- \( \text{urange} \) - \([\text{umin, umax}]\)
- \( \text{vrange} \) - \([\text{vmin, vmax}]\) are lists of numbers with \( \text{umin, umax, vmin, vmax} \)
- \( \text{options} \) - optional string representing plot2d options in gnuplot format

OUTPUT: displays a plot on screen or saves to a file

EXAMPLES:

```sage`
maxima.plot3d_parametric(['\text{v}^\text{sin(u)}', 'v^\text{cos(u)}', 'v'], ['u','v'],[-3.2,3.2],[0,3])  # not tested
opts = ['gnuplot_term, ps'], ['gnuplot_out_file, "sin-cos-plot.eps"']
maxima.plot3d_parametric(['\text{v}^\text{sin(u)}', 'v^\text{cos(u)}', 'v'], ['u','v'],[-3.2,3.2],[0,3],opts)  # not tested
```

The eps file is saved in the current working directory.

Here is a torus:

```sage`
_ = maxima.eval("expr_1: \cos(y)*(10.0+6*\cos(x)); expr_2: \sin(y)*(10.0+6*\cos(x)); expr_3: -6*\sin(x);")
maxima.plot3d_parametric(['expr_1', 'expr_2', 'expr_3'], ['x','y'],[0,6],[0,6])  # not tested
```

Here is a Möbius strip:

```sage`
"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'],[0,6],[0,6])  # not tested
```
\texttt{sage: x = "cos(u)*(3 + v*cos(u/2))"}
\texttt{sage: y = "sin(u)*(3 + v*cos(u/2))"}
\texttt{sage: z = "v*sin(u/2)"}
\texttt{sage: maxima.plot3d_parametric([x,y,z],["u","v"],[-3.1,3.2],[-1/10,1/10]) # not tested}

\textbf{plot\_list}(ptsx, ptsy, options=\texttt{None})

Plots a curve determined by a sequence of points.

\textbf{INPUT:}

- \texttt{ptsx} - [x1,...,xn], where the xi and yi are real,
- \texttt{ptsy} - [y1,...,yn]
- \texttt{options} - a string representing \texttt{maxima plot2d} options.

The points are (x1,y1), (x2,y2), etc.

This function requires \texttt{maxima 5.9.2} or newer.

\textbf{Note:} More that 150 points can sometimes lead to the program hanging. Why?

\textbf{EXAMPLES:}

\texttt{sage: zeta_ptsx = [ (pari(1/2 + i*I/10).zeta().real()).precision(1) for i in range (70,150)]}
\texttt{sage: zeta_ptsy = [ (pari(1/2 + i*I/10).zeta().imag()).precision(1) for i in range (70,150)]}
\texttt{sage: maxima.plot_list(zeta_ptsx, zeta_ptsy) # not tested}
\texttt{sage: opts=["gnuplot_preamble, "set nokey"], ["gnuplot_term, ps"], ["gnuplot_out_file, "zeta.eps"]}
\texttt{sage: maxima.plot_list(zeta_ptsx, zeta_ptsy, opts) # not tested}

\texttt{plot\_multilist}(pts\_list, options=\texttt{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 \texttt{maxima plot2d} options.

\textbf{INPUT:}

- \texttt{pts\_list} - list of points; each point must be of the form [x,y] where x is an integer and y is a real
- \texttt{var} - string; representing Maxima’s plot2d options

Requires \texttt{maxima 5.9.2} at least.

\textbf{Note:} More that 150 points can sometimes lead to the program hanging.

\textbf{EXAMPLES:}

\texttt{sage: xx = [ i/10.0 for i in range (-10,10)]}
\texttt{sage: yy = [ i/10.0 for i in range (-10,10)]}
\texttt{sage: xo = [ 0 for i in range (-10,10)]}
\texttt{sage: yo = [ 0 for i in range (-10,10)]}
\texttt{sage: zeta_ptsx1 = [ (pari(1/2+i*I/10).zeta().real()).precision(1) for i in range (10)]}

(continues on next page)
(continued from previous page)

```python
sage: zeta_ptsy1 = [ (pari(1/2+i*I/10).zeta().imag()).precision(1) for i in range (10) ]
sage: maxima.plot_multilist([[zeta_ptsx1,zeta_ptsy1],[xx,y0],[x0,yy]])
not tested
sage: zeta_ptsx1 = [ (pari(1/2+i*I/10).zeta().real()).precision(1) for i in range (10,150) ]
sage: zeta_ptsy1 = [ (pari(1/2+i*I/10).zeta().imag()).precision(1) for i in range (10,150) ]
sage: maxima.plot_multilist([[zeta_ptsx1,zeta_ptsy1],[xx,y0],[x0,yy]],opts)
not tested
sage: solve_linear(eqns, vars)
Wraps maxima’s linsolve.

INPUT:

- eqns - a list of m strings; each representing a linear question in m = n variables
- vars - a list of n strings; each representing a variable

EXAMPLES:

```python
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]
```

```python
unit_quadratic_integer(n)
Finds a unit of the ring of integers of the quadratic number field \( \mathbb{Q}(\sqrt{n}) \), \( n > 1 \), using the qunit maxima command.

INPUT:

- n - an integer

EXAMPLES:

```python
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?
``` 

```python
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?
``` 

```python
version()
Return the version of Maxima that Sage includes.

INPUT: none

OUTPUT: none

EXAMPLES:
```
sage: maxima.version()  # random
'5.41.0'

class sage.interfaces.maxima_abstract.MaximaAbstractElement(parent, value, is_name=False, name=None)

Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.interface.InterfaceElement

Element of Maxima through an abstract interface.

EXAMPLES:

Elements of this class should not be created directly. The targeted parent of a concrete inherited class should be used instead:

sage: from sage.interfaces.maxima_lib import maxima_lib
sage: xp = maxima(x)
sage: type(xp)
<class 'sage.interfaces.maxima.MaximaElement'>
sage: xl = maxima_lib(x)
sage: type(xl)
<class 'sage.interfaces.maxima_lib.MaximaLibElement'>

comma(args)
Form the expression that would be written ‘self, args’ in Maxima.

INPUT:

  * args - string

OUTPUT: Maxima object

EXAMPLES:

sage: maxima('sqrt(2) + I').comma('numer')
I+1.41421356237309...

comma('a=5')
5*I+sqrt(2)

(continues on next page)

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)

sage: f = maxima('x^2 + 17*y^2')
sage: f.diff('x')
34*y^"x"*diff(y,x,1)+2*x
sage: f.diff('y')
34*y
diff(var='x', n=1)
Return the n-th derivative of self.

INPUT:

• var - variable (default: 'x')
• n - integer (default: 1)

OUTPUT: n-th derivative of self with respect to the variable var

EXAMPLES:

sage: f = maxima('x^2')
sage: f.diff() 2*x
sage: f.diff('x') 2*x
sage: f.diff('x', 2) 2
sage: maxima('sin(x^2)').diff('x', 4)
16*x^4*sin(x^2)-12*sin(x^2)-48*x^2*cos(x^2)

sage: f = maxima('x^2 + 17*y^2')
sage: f.diff('x')
34*y^"x"*diff(y,x,1)+2*x
sage: f.diff('y')
34*y
dot(other)
Implements the notation self . other.

INPUT:

• other - matrix; argument to dot.

OUTPUT: Maxima matrix

EXAMPLES:

sage: A = maxima('matrix ([a1],[a2])')
sage: B = maxima('matrix ([b1, b2])')
sage: A.dot(B)
matrix([[a1*b1, a1*b2], [a2*b1, a2*b2]])

imag()
Return the imaginary part of this Maxima element.

INPUT: none
OUTPUT: Maxima real

EXAMPLES:

```python
sage: maxima('2 + (2/3)*%i').imag()
2/3
```

`integral(var=x, min=None, max=None)`

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

INPUT:
- `var` - variable
- `min` - default: None
- `max` - default: None

OUTPUT:
- the definite integral if xmin is not None
- an indefinite integral otherwise

EXAMPLES:

```python
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
```

`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:

```python
sage: f = maxima('exp(x^2)').integral('x',0,1); f
-(sqrt(%pi)*%i*erf(%i))/2
sage: f.numer()
1.46265174590718...
```
sage: maxima('x^2+1').integral()
x^3/3+x

sage: maxima('x^2+ 1 + y^2').integral('y')
y^3/3+x^2*y+y

sage: maxima('x / (x^2+1)').integral()
log(x^2+1)/2

sage: maxima('1/(x^2+1)').integral()
atan(x)

sage: maxima('1/(x^2+1)').integral('x', 0, infinity)
%pi/2

sage: maxima('x/(x^2+1)').integral('x', -1, 1)
0

sage: f = maxima('exp(x^2)').integral('x',0,1); f
-(sqrt(%pi)*%i*erf(%i))/2

sage: f.numer()
1.46265174590718...

nintegral(var='x', a=0, b=1, desired_relative_error='1e-8', maximum_num_subintervals=200)
Return a numerical approximation to the integral of self from a to b.

INPUT:
• var - variable to integrate with respect to
• a - lower endpoint of integration
• b - upper endpoint of integration
• desired_relative_error - (default: '1e-8') the desired relative error
• maximum_num_subintervals - (default: 200) maxima number of subintervals

OUTPUT:
• approximation to the integral
• estimated absolute error of the approximation
• the number of integrand evaluations
• an error code:
  – 0 - no problems were encountered
  – 1 - too many subintervals were done
  – 2 - excessive roundoff error
  – 3 - extremely bad integrand behavior
  – 4 - failed to converge
  – 5 - integral is probably divergent or slowly convergent
  – 6 - the input is invalid

EXAMPLES:

sage: maxima('exp(-sqrt(x))').nintegral('x',0,1)
(0.5284822353142306, 4.1633141378838...e-11, 231, 0)

Note that GP also does numerical integration, and can do so to very high precision very quickly:
numer()
Return numerical approximation to self as a Maxima object.

INPUT: none

OUTPUT: Maxima object

EXAMPLES:

```
sage: a = maxima('sqrt(2)').numer(); a
1.41421356237309...
sage: type(a)
<class 'sage.interfaces.maxima.MaximaElement'>
```

partial_fraction_decomposition(var='x')
Return the partial fraction decomposition of self with respect to the variable var.

INPUT:

• var - string

OUTPUT: Maxima object

EXAMPLES:

```
sage: f = maxima('1/((1+x)*(x-1))')
sage: f.partial_fraction_decomposition('x')
1/(2*(x-1))-1/(2*(x+1))
sage: print(f.partial_fraction_decomposition('x'))
1 1
--------- - ---------
2 (x - 1) 2 (x + 1)
```

real()
Return the real part of this Maxima element.

INPUT: none

OUTPUT: Maxima real

EXAMPLES:

```
sage: maxima('2 + (2/3)*%i').real()
2
```

str()
Return string representation of this Maxima object.

INPUT: none

OUTPUT: string

EXAMPLES:
sage: maxima('sqrt(2) + 1/3').str()
'sqrt(2)+1/3'

**subst(val)**

Substitute a value or several values into this Maxima object.

**INPUT:**

- val - string representing substitution(s) to perform

**OUTPUT:** Maxima object

**EXAMPLES:**

```python
sage: maxima('a^2 + 3*a + b').subst('b=2')
a^2+3*a+2
sage: maxima('a^2 + 3*a + b').subst('a=17')
b+340
sage: maxima('a^2 + 3*a + b').subst('a=17, b=2')
342
```

**class** `sage.interfaces.maxima_abstract.MaximaAbstractElementFunction(parent, name, defn, args, latex)`

**Bases:** `sage.interfaces.maxima_abstract.MaximaAbstractElement`

Create a Maxima function with the parent `parent`, name `name`, definition `defn`, arguments `args` and latex representation `latex`.

**INPUT:**

- parent - an instance of a concrete Maxima interface
- name - string
- defn - string
- args - string; comma separated names of arguments
- latex - string

**OUTPUT:** Maxima function

**EXAMPLES:**

```python
sage: maxima.function('x,y','e^cos(x)')
e^cos(x)
```

**arguments(split=True)**

Returns the arguments of this Maxima function.

**INPUT:**

- split - boolean; if True return a tuple of strings, otherwise return a string of comma-separated arguments

**OUTPUT:**

- a string if split is False
- a list of strings if split is True

**EXAMPLES:**
sage: f = maxima.function(['x','y'],'sin(x+y)')
sage: f.arguments()
['x', 'y']
sage: f.arguments(split=False)
'x,y'
sage: f = maxima.function('', 'sin(x)')
sage: f.arguments()
[]

**definition()**

Returns the definition of this Maxima function as a string.

**INPUT:** none

**OUTPUT:** string

**EXAMPLES:**

```python
sage: f = maxima.function(['x','y'],'sin(x+y)')
sage: f.definition()
'sin(x+y)'
```

**integral**(var)

Returns the integral of self with respect to the variable var.

**INPUT:**

- var - a variable

**OUTPUT:** Maxima function

Note that integrate is an alias of integral.

**EXAMPLES:**

```python
sage: x,y = var('x,y')
sage: f = maxima.function('x','sin(x)')
sage: f.integral(x)
-cos(x)
sage: f.integral(y)
sin(x)*y
```

**integrate**(var)

Returns the integral of self with respect to the variable var.

**INPUT:**

- var - a variable

**OUTPUT:** Maxima function

Note that integrate is an alias of integral.

**EXAMPLES:**

```python
sage: x,y = var('x,y')
sage: f = maxima.function('x','sin(x)')
sage: f.integrate(x)
-cos(x)
```
sage: f.integral(y)
sin(x)*y

sage.interfaces.maxima_abstract.maxima_console()
Spawn a new Maxima command-line session.

EXAMPLES:

sage: from sage.interfaces.maxima_abstract import maxima_console
sage: maxima_console()
# not tested
Maxima 5.34.1 http://maxima.sourceforge.net
...

sage.interfaces.maxima_abstract.maxima_version()
Return Maxima version.

Currently this calls a new copy of Maxima.

EXAMPLES:

sage: from sage.interfaces.maxima_abstract import maxima_version
sage: maxima_version()
# random
'5.41.0'

sage.interfaces.maxima_abstract.reduce_load_MaximaAbstract_function(parent, defn, args, latex)
Unpickle a Maxima function.

EXAMPLES:

sage: from sage.interfaces.maxima_abstract import reduce_load_MaximaAbstract_function
sage: f = maxima.function('x,y','sin(x+y)')
sage: _,args = f.__reduce__()
sage: g = reduce_load_MaximaAbstract_function(*args)
sage: g == f
True
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

For this interface, Maxima is loaded into ECL which is itself loaded as a C library in Sage. Translations between Sage and Maxima objects (which are nothing but wrappers to ECL objects) is made as much as possible directly, but falls back to the string based conversion used by the classical Maxima Pexpect interface in case no new implementation has been made.

This interface is the one used for calculus by Sage and is accessible as `maxima_calculus`:

```
sage: maxima_calculus
Maxima_lib
```

Only one instance of this interface can be instantiated, so the user should not try to instantiate another one, which is anyway set to raise an error:

```
sage: from sage.interfaces.maxima_lib import MaximaLib
sage: MaximaLib()
Traceback (most recent call last):
...
RuntimeError: Maxima interface in library mode can only be instantiated once
```

Changed `besselexpand` to true in `init_code` – automatically simplify Bessel functions to trig functions when appropriate when true. Examples:

For some infinite sums, a closed expression can be found. By default, “maxima” is used for that:
Maxima has some flags that affect how the result gets simplified (By default, besselexpand is false in Maxima; however in 5.39 this test does not show any difference, as, apparently, another expansion path is used):

```
sage: maxima_calculus("besselexpand:false")
false
sage: x,n,k = var("x","n","k")
sage: sum((-x)^n/(factorial(n)*factorial(n+3/2)),n,0,oo)
-1/2*(2*x*cos(2*sqrt(x)) - sqrt(x)*sin(2*sqrt(x)))/(sqrt(pi)*x^2)
sage: maxima_calculus("besselexpand:true")
true
```

The output is parseable (i.e. trac ticket #31796 is fixed):

```
sage: foo = maxima_calculus('a and (b or c)') ; foo
a and (b or c)
sage: bar = maxima_calculus(foo) ; bar
a and (b or c)
sage: bar == foo
True
```

```
class sage.interfaces.maxima_lib.MaximaLib
    Bases: sage.interfaces.maxima_abstract.MaximaAbstract

    Interface to Maxima as a Library.

    INPUT: none

    OUTPUT: Maxima interface as a Library

    EXAMPLES:

    sage: from sage.interfaces.maxima_lib import MaximaLib, maxima_lib
    sage: isinstance(maxima_lib,MaximaLib)
    True

    Only one such interface can be instantiated:

    sage: MaximaLib()
    Traceback (most recent call last):
    ...
    RuntimeError: Maxima interface in library mode can only be instantiated once

    clear(var)
    Clear the variable named var.

    INPUT:
      • var - string

    OUTPUT: none

    EXAMPLES:
```
eval(line, locals=None, reformat=True, **kwds)
Evaluate the line in Maxima.

INPUT:

- line - string; text to evaluate
- locals - None (ignored); this is used for compatibility with the Sage notebook’s generic system interface.
- reformat - boolean; whether to strip output or not
- **kwds - All other arguments are currently ignored.

OUTPUT: string representing Maxima output

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib._eval_line('1+1')
'2'
sage: maxima_lib._eval_line('1+1;')
'2'
sage: maxima_lib._eval_line('1+1$')
''
sage: maxima_lib._eval_line('randvar : cos(x)+sin(y)$')
''
sage: maxima_lib._eval_line('randvar')
'sin(y)+cos(x)'
```

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

INPUT:

- var - string

OUTPUT: string

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib.set('xxxxx', '2')
sage: maxima_lib.get('xxxxx')
'2'
```

lisp(cmd)
Send a lisp command to maxima.

INPUT:

- cmd - string
OUTPUT: ECL object

Note:  The output of this command is very raw - not pretty.

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib.lisp("(+ 2 17)"
<ECL: 19>
```

`set(var, value)`
Set the variable var to the given value.

**INPUT:**
- `var` - string
- `value` - string

**OUTPUT:** none

**EXAMPLES:**

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib.set('xxxxx', '2')
sage: maxima_lib.get('xxxxx')
'2'
```

`sr_integral(*args)`
Helper function to wrap calculus use of Maxima's integration.

`sr_limit(expr, v, a, dir=None)`
Helper function to wrap calculus use of Maxima's limits.

`sr_prod(*args)`
Helper function to wrap calculus use of Maxima's product.

`sr_sum(*args)`
Helper function to wrap calculus use of Maxima's summation.

`sr_tlimit(expr, v, a, dir=None)`
Helper function to wrap calculus use of Maxima's Taylor series limits.

```
class sage.interfaces.maxima_lib.MaximalLibElement(parent, value, is_name=False, name=None)
Bases: sage.interfaces.maxima_abstract.MaximaAbstractElement
Element of Maxima through library interface.

EXAMPLES:

Elements of this class should not be created directly. The targeted parent should be used instead:

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib(4)
4
sage: maxima_lib(log(x))
log(_SAGE_VAR_x)
```

`display2d(onscreen=True)`
Return the 2d representation of this Maxima object.
INPUT:
  • onscreen - boolean (default: True); whether to print or return

OUTPUT:
The representation is printed if onscreen is set to True and returned as a string otherwise.

EXAMPLES:

```python
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: F = maxima_lib('x^5 - y^5').factor()
sage: F.display2d()
    4 3 2 2 3 4
   - (y - x) (y + x y + x y + x y + x )
```

ecl()
Return the underlying ECL object of this MaximaLib object.

INPUT: none

OUTPUT: ECL object

EXAMPLES:

```python
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib(x+cos(19)).ecl()
<ECL: ((MPLUS SIMP) ((%COS SIMP) 19) |$_SAGE_VAR_x|)>
```

to_poly_solve(vars, options="")
Use Maxima’s to_poly_solver package.

INPUT:
  • vars - symbolic expressions
  • options - string (default="")

OUTPUT: Maxima object

EXAMPLES:
The zXXX below are names for arbitrary integers and subject to change:

```python
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: sol = maxima_lib(sin(x) == 0).to_poly_solve(x)
sage: sol.sage()
[[x == pi*z...]]
```

class sage.interfaces.maxima_lib.MaximaLibElementFunction(parent, name, defn, args, latex)

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')

\texttt{f.ecl()}

\texttt{<ECL: ((\%INTEGRATE SIMP) \((\$F SIMP) \$X) \$X)>}

\texttt{sage: dummy_integrate(f.ecl())}

\texttt{integrate(f(x), x)}

\texttt{sage: f = maxima_lib('f(x)').integrate('x',0,10)}

\texttt{sage: f.ecl()}

\texttt{<ECL: ((\%INTEGRATE SIMP) \((\$F SIMP) \$X) \$X \ 0 \ 10)>}

\texttt{sage: dummy_integrate(f.ecl())}

\texttt{integrate(f(x), x, 0, 10)}

\texttt{sage.interfaces.maxima_lib.is_MaximaLibElement(x)}

Returns True if x is of type MaximaLibElement.

\textbf{EXAMPLES:}

\texttt{sage: from sage.interfaces.maxima_lib import maxima_lib, is_MaximaLibElement}

\texttt{sage: m = maxima_lib(1)}

\texttt{sage: is(MaximaLibElement(m))}

\texttt{True}

\texttt{sage: is(MaximaLibElement(1))}

\texttt{False}

\texttt{sage.interfaces.maxima_lib.max_at_to_sage(expr)}

Special conversion rule for AT expressions.

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{expr} - ECL object; a Maxima AT expression
\end{itemize}

\textbf{OUTPUT:} symbolic expression

\textbf{EXAMPLES:}

\texttt{sage: from sage.interfaces.maxima_lib import maxima_lib, max_at_to_sage}

\texttt{sage: a=maxima_lib("at(f(x,y,z),[x=1,y=2,z=3])")}

\texttt{sage: a}

\texttt{at(f(x, y, z), [x = 1, y = 2, z = 3])}

\texttt{sage: max_at_to_sage(a.ecl())}

\texttt{f(1, 2, 3)}

\texttt{sage: a=maxima_lib("'at(f(x,y,z),x=1)"")}

\texttt{sage: a}

\texttt{'at(f(x, y, z), x = 1)}

\texttt{sage: max_at_to_sage(a.ecl())}

\texttt{f(1, y, z)}

\texttt{sage.interfaces.maxima_lib.max_harmonic_to_sage(expr)}

\textbf{EXAMPLES:}

\texttt{sage: from sage.interfaces.maxima_lib import maxima_lib, max_to_sr}

\texttt{sage: c=maxima_lib(harmonic_number(x,2))}

\texttt{sage: c.ecl()}

\texttt{<ECL: ((\$GEN_HARMONIC_NUMBER SIMP) \(2 \ |$_SAGE_VAR_x|)>}

(continues on next page)
sage: max_to_sr(c.ecl())
harmonic_number(x, 2)

`sage.interfaces.maxima_lib.max_to_sr(expr)`
Convert a Maxima object into a symbolic expression.

**INPUT:**
- `expr` - ECL object

**OUTPUT:** symbolic expression

**EXAMPLES:**

```python
sage: from sage.interfaces.maxima_lib import maxima_lib, max_to_sr
sage: f = maxima_lib('f(x)')
```

```python
sage: f.ecl()
<ECL: (($F SIMP) $X)>
```

```python
sage: max_to_sr(f.ecl())
f(x)
```

`sage.interfaces.maxima_lib.max_to_string(s)`
Return the Maxima string corresponding to this ECL object.

**INPUT:**
- `s` - ECL object

**OUTPUT:** string

**EXAMPLES:**

```python
sage: from sage.interfaces.maxima_lib import maxima_lib, max_to_string
sage: ecl = maxima_lib(cos(x)).ecl()
```

```python
sage: max_to_string(ecl)
'cos(_SAGE_VAR_x)'
```

`sage.interfaces.maxima_lib.mdiff_to_sage(expr)`
Special conversion rule for %DERIVATIVE expressions.

**INPUT:**
- `expr` - ECL object; a Maxima %DERIVATIVE expression

**OUTPUT:** symbolic expression

**EXAMPLES:**

```python
sage: from sage.interfaces.maxima_lib import maxima_lib, mdiff_to_sage
sage: f = maxima_lib('f(x)').diff('x',4)
```

```python
sage: f.ecl()
<ECL: (%DERIVATIVE SIMP) (($F SIMP) $X) $X 4)>
```

```python
sage: mdiff_to_sage(f.ecl())
diff(f(x), x, x, x, x)
```

`sage.interfaces.maxima_lib.mlist_to_sage(expr)`
Special conversion rule for MLIST expressions.

**INPUT:**
• **expr** - ECL object; a Maxima MLIST expression (i.e., a list)

OUTPUT: a Python list of converted expressions.

EXAMPLES:

```python
sage: from sage.interfaces.maxima_lib import maxima_lib, mlist_to_sage
sage: L = maxima_lib('[1,2,3]')
sage: L.ecl()
<ECL: ((MLIST SIMP) 1 2 3)>
sage: mlist_to_sage(L.ecl())
[1, 2, 3]
```

`sage.interfaces.maxima_lib.mqapply_to_sage(expr)`  
Special conversion rule for MQAPPLY expressions.

INPUT:

• **expr** - ECL object; a Maxima MQAPPLY expression

OUTPUT: symbolic expression

MQAPPLY is used for function as li[x](y) and psi[x](y).

EXAMPLES:

```python
sage: from sage.interfaces.maxima_lib import maxima_lib, mqapply_to_sage
sage: c = maxima_lib('li[2](3)')
sage: c.ecl()
<ECL: ((MQAPPLY SIMP) (($LI SIMP ARRAY) 2) 3)>
sage: mqapply_to_sage(c.ecl())
dilog(3)
```

`sage.interfaces.maxima_lib.mrat_to_sage(expr)`  
Convert a Maxima MRAT expression to Sage SR.

INPUT:

• **expr** - ECL object; a Maxima MRAT expression

OUTPUT: symbolic expression

Maxima has an optimised representation for multivariate rational expressions. The easiest way to translate those to SR is by first asking Maxima to give the generic representation of the object. That is what RATDISREP does in Maxima.

EXAMPLES:

```python
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_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^3 + x*y)/(x^6*y)
```
sage.interfaces.maxima_lib.parse_max_string(s)
Evaluate string in Maxima without any further simplification.

INPUT:
• s - string

OUTPUT: ECL object

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import parse_max_string
sage: parse_max_string('1+1')
<ECL: ((MPLUS) 1 1)>
```

sage.interfaces.maxima_lib.pyobject_to_max(obj)
Convert a (simple) Python object into a Maxima object.

INPUT:
• expr - Python object

OUTPUT: ECL object

Note: This uses functions defined in sage.libs.ecl.

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import pyobject_to_max
sage: pyobject_to_max(4)
<ECL: 4>
sage: pyobject_to_max('z')
<ECL: Z>
sage: var('x')
x
sage: pyobject_to_max(x)
Traceback (most recent call last):
...TypeError: Unimplemented type for python_to_ecl
```

sage.interfaces.maxima_lib.reduce_load_MaximaLib()
Unpickle the (unique) Maxima library interface.

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import reduce_load_MaximaLib
sage: reduce_load_MaximaLib()
Maxima_lib
```

sage.interfaces.maxima_lib.sage_rat(x, y)
Return quotient x/y.

INPUT:
• x - integer
• y - integer

OUTPUT: rational
EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import sage_rat
sage: sage_rat(1,7)
1/7
```

`sage.interfaces.maxima_lib.sr_to_max(expr)`  
Convert a symbolic expression into a Maxima object.

**INPUT:**  
- `expr` - symbolic expression

**OUTPUT:** ECL object

**EXAMPLES:**

```
sage: from sage.interfaces.maxima_lib import sr_to_max
sage: var('x')
x
sage: sr_to_max(x)
<ECL: $X>
sage: sr_to_max(cos(x))
<ECL: ((%COS) $X)>
sage: f = function('f')(x)
sage: sr_to_max(f.diff())
<ECL: ((%DERIVATIVE) ((F) $X) $X 1)>
```

`sage.interfaces.maxima_lib.stdout_to_string(s)`  
Evaluate command `s` and catch Maxima stdout (not the result of the command!) into a string.

**INPUT:**  
- `s` - string; command to evaluate

**OUTPUT:** string

This is currently used to implement `display2d()`.

**EXAMPLES:**

```
sage: from sage.interfaces.maxima_lib import stdout_to_string
sage: stdout_to_string('1+1')
''
sage: stdout_to_string('disp(1+1)')
'2\n\n'
```
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.

```python
class sage.interfaces.mupad.Mupad(maxread=None, script_subdirectory=None, server=None, server_tmpdir=None, logfile=None):
    # Interface to the MuPAD interpreter.

completions(string, strip=False)

EXAMPLES:

```
sage: mupad.completions('linal') # optional - mupad
['linalg']
```

```python
call()
```

Spawn a new MuPAD command-line session.

EXAMPLES:

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

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

EXAMPLES:

```
sage: t = mupad.cputime()
```

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

EXAMPLES:

```
sage: t = mupad.eval('0.1160000000000001') #random, optional - MuPAD
0.1160000000000001
```

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

expect()

EXAMPLES:

sage: a = mupad(1)  # optional - mupad
sage: mupad.expect()  # optional - mupad
<pexpect.spawn instance at 0x...>

get(var)

Get the value of the variable var.

EXAMPLES:

sage: mupad.set('a', 4)  # optional - mupad
sage: mupad.get('a').strip()  # optional - mupad
'4'

set(var, value)

Set the variable var to the given value.

EXAMPLES:

sage: mupad.set('a', 4)  # optional - mupad
sage: mupad.get('a').strip()  # optional - mupad
'4'

class sage.interfaces.mupad.MupadElement(parent, value, is_name=False, name=None)

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

class sage.interfaces.mupad.MupadFunction(parent, name)

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

class sage.interfaces.mupad.MupadFunctionElement(obj, name)

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

sage.interfaces.mupad.mupad_console()

Spawn a new MuPAD command-line session.

EXAMPLES:

sage: from sage.interfaces.mupad import mupad_console
sage: mupad_console()  # not tested

*----* MuPAD Pro 4.0.2 -- The Open Computer Algebra System
/| /|
*----* | Copyright (c) 1997 - 2007 by SciFace Software
| *--|--* All rights reserved.
|/ |/
*----* Licensed to: ...

sage.interfaces.mupad.reduce_load_mupad()

EXAMPLES:
<table>
<thead>
<tr>
<th>sage:</th>
<th>from sage.interfaces.mupad import reduce_load_mupad</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage:</td>
<td>reduce_load_mupad()</td>
</tr>
<tr>
<td></td>
<td>Mupad</td>
</tr>
</tbody>
</table>
CHAPTER
THIRTYTWO

INTERFACE TO MWRANK

\texttt{sage.interfaces.mwrank.Mwrank}(\texttt{options=", server=None, server_tmpdir=None})

Create and return an mwrank interpreter, with given options.

INPUT:

- **options** - string; passed when starting mwrank. The format is:

  -h help prints this info and quits
  -q quiet turns OFF banner display and prompt
  -v n verbosity sets verbosity to n (default=1)
  -o PARi/GP output turns ON extra PARi/GP short output (default is OFF)
  -p n precision sets precision to n decimals (default=15)
  -b n quartic bound bound on quartic point search (default=10)
  -x n n aux number of aux primes used for sieving (default=6)
  -l list turns ON listing of points (default ON unless v=0)
  -s selmer_only if set, computes Selmer rank only (default: not set)
  -d skip_2nd_descent if set, skips the second descent for curves with 2-torsion (default: not set)
  -S n sat_bd upper bound on saturation primes (default=100, -1 for automatic)

**Warning:** Do not use the option "-q" which turns off the prompt.

EXAMPLES:

```
sage: M = Mwrank('-v 0 -l')
sage: print(M('0 0 1 -1 0'))
Curve [0,0,1,-1,0] : Rank = 1
Generator 1 is [0:-1:1]; height 0.051...
Regulator = 0.051...
```

\texttt{class sage.interfaces.mwrank.Mwrank_class}(\texttt{options=", server=None, server_tmpdir=None})

Bases: \texttt{sage.interfaces.expect.Expect}

Interface to the Mwrank interpreter.

\texttt{console()}

Start the mwrank console.

EXAMPLES:
eval\((s, \texttt{**kwds})\)
Return mwrank’s output for the given input.

**INPUT:**

- \(s\) (str) - 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 9.5 Reference Manual: Interpreter Interfaces, Release 9.5

```python
sage: from sage.interfaces.mwrank import validate_mwrank_input
sage: validate_mwrank_input([1,2,3,4,5])
'[1, 2, 3, 4, 5]'
sage: validate_mwrank_input([-1,2,-3,4,-55])
'[-1, 2, -3, 4, -55]'
sage: validate_mwrank_input([1,2,3,4])
Traceback (most recent call last):
... ValueError: [1, 2, 3, 4] is not valid input to mwrank (should have 5 entries)
sage: validate_mwrank_input([1,2,3,4,i])
Traceback (most recent call last):
... ValueError: [1, 2, 3, 4, I] is not valid input to mwrank (entries should be integers)
```

A string of the form `[a1,a2,a3,a4,a6]` with any whitespace and integers ai:

```python
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
```
GNU Octave is a free software (GPL) MATLAB-like program with numerical routines for integrating, solving systems of equations, special functions, and solving (numerically) differential equations. Please see http://octave.org/ for more details.

The commands in this section only work if you have the optional “octave” interpreter installed and available in your PATH. It’s not necessary to install any special Sage packages.

EXAMPLES:

```
sage: octave.eval('2+2')  # optional - octave
'ans = 4'
sage: a = octave(10)     # optional - octave
sage: a**10              # optional - octave
1e+10
```

LOG: - creation (William Stein) - ? (David Joyner, 2005-12-18) - Examples (David Joyner, 2005-01-03)

### 33.1 Computation of Special Functions

Octave implements computation of the following special functions (see the maxima and gp interfaces for even more special functions):

- **airy**
  - Airy functions of the first and second kind, and their derivatives.
  
  \[
  \text{airy}(0,x) = Ai(x), \quad \text{airy}(1,x) = Ai'(x), \quad \text{airy}(2,x) = Bi(x), \quad \text{airy}(3,x) = Bi'(x)
  \]

- **besselj**
  - Bessel functions of the first kind.

- **bessely**
  - Bessel functions of the second kind.

- **besseli**
  - Modified Bessel functions of the first kind.

- **besselk**
  - Modified Bessel functions of the second kind.

- **besselh**
  - Compute Hankel functions of the first \((k = 1)\) or second \((k = 2)\) kind.

- **beta**
  - The Beta function,
  
  \[
  \beta(a, b) = \frac{\Gamma(a) \cdot \Gamma(b)}{\Gamma(a + b)}
  \]

(continues on next page)
The incomplete Beta function, \( \text{erf} \)
The error function, \( \text{erfinv} \)
The inverse of the error function.
\( \text{gamma} \)
The Gamma function, \( \text{gammainc} \)
The incomplete gamma function,

For example,

\begin{verbatim}
  sage: octave("airy(3,2)"")  # optional - octave
  4.10068
  sage: octave("beta(2,2)"")  # optional - octave
  0.166667
  sage: octave("betainc(0.2,2,2)"")  # optional - octave
  0.104
  sage: octave("besselh(0,2)"")  # optional - octave
  (0.223891, 0.510376)
  sage: octave("besselh(0,1)"")  # optional - octave
  (0.765198, 0.088257)
  sage: octave("besseli(1,2)"")  # optional - octave
  1.59064
  sage: octave("besselj(1,2)"")  # optional - octave
  0.576725
  sage: octave("besselk(1,2)"")  # optional - octave
  0.139866
  sage: octave("erf(0)"")  # optional - octave
  0
  sage: octave("erf(1)"")  # optional - octave
  0.842701
  sage: octave("erfinv(0.842)"")  # optional - octave
  0.998315
  sage: octave("gamma(1.5)"")  # optional - octave
  0.886227
  sage: octave("gammainc(1.5,1)"")  # optional - octave
  0.77687
\end{verbatim}

33.2 Tutorial

EXAMPLES:

\begin{verbatim}
  sage: octave("4+10")  # optional - octave
  14
  sage: octave("date")  # optional - octave; random output
  18-Oct-2007
  sage: octave("5*10 + 6")  # optional - octave
  56
  sage: octave("(6+6)/3")  # optional - octave
\end{verbatim}
Sage 9.5 Reference Manual: Interpreter Interfaces, Release 9.5

(continued from previous page)

4
sage: octave('9')^2  # optional - octave
81
sage: a = octave(10); b = octave(20); c = octave(30)  # optional - octave
sage: avg = (a+b+c)/3  # optional - octave
20
sage: parent(avg)  # optional - octave
Octave

sage: my_scalar = octave('3.1415')  # optional - octave
3.1415
sage: my_vector1 = octave('[1,5,7]')  # optional - octave
1 5 7
sage: my_vector2 = octave('[1;5;7]')  # optional - octave
1
5
7
sage: my_vector1 * my_vector2  # optional - octave
75

class sage.interfaces.octave.Octave(maxread=None, script_subdirectory=None, logfile=None, server=None, server_tmpdir=None, seed=None, command=None)

Bases: sage.interfaces.expect.Expect

Interface to the Octave interpreter.

EXAMPLES:

sage: octave.eval("a = [ 1, 1, 2; 3, 5, 8; 13, 21, 33 ]")  # optional - octave
'a =

 1 1 2
 3 5 8
 13 21 33

'sage: octave.eval("b = [ 1; 3; 13]")  # optional - octave
'b =

 1
 3
 13

'sage: octave.eval("c=a \b")  # solves linear equation: a\cdot c = b  # optional ...
→ octave; random output
'c =

 1 7.21645e-16 -7.21645e-16

'sage: octave.eval("c")  # optional - octave; random...
→ output
'c =

 1 7.21645e-16 -7.21645e-16


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"

33.2. Tutorial 267
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:

```python
sage: octave_console()  # not tested
GNU Octave, version 2.1.73 (i386-apple-darwin8.5.3).
Copyright (C) 2006 John W. Eaton.
...
octave:1> 2+3
ans = 5
octave:2> [ctl-d]
```

Pressing ctrl-d exits the octave console and returns you to Sage. octave, like Sage, remembers its history from one session to another.

de_system_plot(f, ics, trange)

Plot (using octave’s interface to gnuplot) the solution to a $2 \times 2$ 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:

```python
sage: octave.de_system_plot(['x+y', 'x-y'], [1,-1], [0,2])  # not tested -- does...
\rightarrow this actually work (on OS X it fails for me -- William Stein, 2007-10)
```

This should yield the two plots $(t, x(t)), (t, y(t))$ on the same graph (the $t$-axis is the horizontal axis) of the system of ODEs

$$
x' = x + y, x(0) = 1; \quad y' = x - y, y(0) = -1, \quad \text{for} \quad 0 < t < 2.
$$

get(var)

Get the value of the variable var.

EXAMPLES:

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

quit(verbos=False)

EXAMPLES:

```python
sage: o = Octave()
sage: o._start()  # optional - octave
sage: o.quit(True)  # optional - octave
Exiting spawned Octave process.
```
**sage2octave_matrix_string**(*A*)

Return an octave matrix from a Sage matrix.

**INPUT:** A Sage matrix with entries in the rationals or reals.

**OUTPUT:** A string that evaluates to an Octave matrix.

**EXAMPLES:**

```python
sage: M33 = MatrixSpace(QQ,3,3)
sage: A = M33([1,2,3,4,5,6,7,8,0])
sage: octave.sage2octave_matrix_string(A)  # optional - octave
'[[1, 2, 3; 4, 5, 6; 7, 8, 0]]
```

**AUTHORS:**

- David Joyner and William Stein

**set**(*var*, *value*)

Set the variable *var* to the given *value*.

**EXAMPLES:**

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

**set_seed**(*seed=None*)

Set the seed for the random number generator for this octave interpreter.

**EXAMPLES:**

```python
sage: o = Octave()  # optional - octave
sage: o.set_seed(1)  # optional - octave
1
sage: [o.rand() for i in range(5)]  # optional - octave
[0.134364, 0.847434, 0.763775, 0.255069, 0.495435]
```

**solve_linear_system**(*A*, *b*)

Use octave to compute a solution *x* to *A*x = *b*, as a list.

**INPUT:**

- *A* – mxn matrix *A* with entries in *Q* or *R*
- *b* – m-vector *b* entries in *Q* or *R* (resp)

**OUTPUT:** A list *x* (if it exists) which solves *M*x = *b*

**EXAMPLES:**

```python
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)
[-0.3333329999999999, 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
```

class sage.interfaces.octave.OctaveElement (parent, value, is_name=False, name=None)

Bases: sage.interfaces.expect.ExpectElement

sage.interfaces.octave.octave_console()

Spawn a new Octave command-line session.

This requires that the optional octave program be installed and in your PATH, but no optional Sage packages need be installed.

EXAMPLES:

```
sage: octave_console() # not tested
GNU Octave, version 2.1.73 (i386-apple-darwin8.5.3).
Copyright (C) 2006 John W. Eaton.
...
octave:1> 2+3
ans = 5
octave:2> [ctl-d]
```

Pressing ctrl-d exits the octave console and returns you to Sage. octave, like Sage, remembers its history from one session to another.

sage.interfaces.octave.reduce_load_Octave()

EXAMPLES:

```
sage: from sage.interfaces.octave import reduce_load_Octave
sage: reduce_load_Octave()
Octave
```

sage.interfaces.octave.to_complex(octave_string, R)

Helper function to convert octave complex number
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:

```python
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...tests] for sol in v.solution_dicts()]  # optional -- phc
sage: residuals  # optional -- phc
[[[0, 0], [0, 0]]
```

blackbox(polys, input_ring, verbose=False)

Return as a string the result of running PHC with the given polynomials under blackbox mode (the ‘-b’ option).
INPUT:

• polys – a list of multivariate polynomials (elements of a multivariate polynomial ring).
• input_ring – for coercion of the variables into the desired ring.
• verbose – print lots of verbose information about what this function does.

OUTPUT:

• a PHC_Object object containing the phcpack output string.

EXAMPLES:

```python
sage: from sage.interfaces.phc import *
sage: R2.<x,y> = PolynomialRing(QQ,2)
sage: start_sys = [x^6-y^2,y^5-1] # optional -- phc
sage: sol = phc.blackbox(start_sys, R2)
```

### mixed_volume

Computes the mixed volume of the polynomial system given by the input polys.

INPUT:

• polys – a list of multivariate polynomials (elements of a multivariate polynomial ring).
• verbose – print lots of verbose information about what this function does.

OUTPUT:

• The mixed volume.

EXAMPLES:

```python
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] # optional -- phc
sage: phc.mixed_volume(test_sys)
```

### path_track

This function computes homotopy paths between the solutions of start_sys and end_sys.

INPUT:

• start_sys – a square polynomial system, given as a list of polynomials
• end_sys – same type as start_sys
• input_ring – for coercion of the variables into the desired ring.
• c_skew – optional, the imaginary part of homotopy multiplier; nonzero values are often necessary to avoid intermediate path collisions
• saved_start – optional. A phc output file. If not given, start system solutions are computed via the phc.blackbox function.

OUTPUT:

• a list of paths as dictionaries, with the keys variables and t-values on the path.

EXAMPLES:
```python
from sage.interfaces.phc import *
R2.<x,y> = PolynomialRing(QQ,2)
start_sys = [x^6-y^2,y^5-1]
sol = phc.blackbox(start_sys, R2)  # optional -- phc
start_save = sol.save_as_start()  # optional -- phc
end_sys = [x^7-2,y^5-x^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
```

```python
from sage.interfaces.phc import *
from sage.structure.sage_object import SageObject
R2.<x,y> = PolynomialRing(QQ,2)
start_sys = [x^5-y^2,y^5-1]
sol = phc.blackbox(start_sys, R2)  # optional -- phc
start_save = sol.save_as_start()  # optional -- phc
end_sys = [x^5-25,y^5-x^2]  # optional -- phc
testing = phc.plot_paths_2d(start_sys, end_sys, R2)  # optional -- phc
```

```python
from sage.interfaces.phc import *
from sage.structure.sage_object import SageObject
R2.<x,y> = PolynomialRing(QQ,2)
start_sys = [x^5-y^2,y^5-1]
sol = phc.blackbox(start_sys, R2)  # optional -- phc
start_save = sol.save_as_start()  # optional -- phc
end_sys = [x^5-25,y^5-x^2]  # 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'>
```

```python
from sage.interfaces.phc import *
from sage.structure.sage_object import SageObject
R2.<x,y> = PolynomialRing(QQ,2)
start_sys = [x^5-y^2,y^5-1]
sol = phc.blackbox(start_sys, R2)  # optional -- phc
start_save = sol.save_as_start()  # optional -- phc
end_sys = [x^5-25,y^5-x^2]  # optional -- phc
testing = phc.plot_paths_2d(start_sys, end_sys, R2)  # 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'>
```

plot_paths_2d(start_sys, end_sys, input_ring, c_skew=0.001, endpoints=True, saved_start=None, rand_colors=False)
This returns 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:

```python
from sage.interfaces.phc import *
from sage.structure.sage_object import SageObject
R2.<x,y> = PolynomialRing(QQ,2)
start_sys = [x^5-y^2,y^5-1]
sol = phc.blackbox(start_sys, R2)  # optional -- phc
start_save = sol.save_as_start()  # optional -- phc
end_sys = [x^5-25,y^5-x^2]  # 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 – a list of multivariate polynomials (elements of a multivariate polynomial ring).
- input_ring: for coercion of the variables into the desired ring.
• path_track_file: whether to save path-tracking information
• verbose – print lots of verbose information about what this function does.

OUTPUT:
• A solution in the form of a PHCObject.

EXAMPLES:
```python
sage: from sage.interfaces.phc import *
sage: R2.<x,y> = PolynomialRing(QQ,2)
sage: start_sys = [x^6-y^2,y^5-1]
```

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:
```python
sage: from sage.interfaces.phc import phc
sage: R.<x,y> = PolynomialRing(CC,2)
sage: p_sys = [x^10-y,y^2-1]
```

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:
```python
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'

save_as_start(start_filename=None, sol_filter="")
Saves a solution as a phcpack start file. The usual output is just as a string, but it can be saved to a file as well. Even if saved to a file, it still returns the output string.

EXAMPLES:

sage: from sage.interfaces.phc import phc
sage: R2.<x,y> = PolynomialRing(QQ,2)
sage: start_sys = [x^3-y^2,y^5-1]
sage: sol = phc.blackbox(start_sys, R2)  # optional -- phc
sage: start_save = sol.save_as_start()  # optional -- phc
sage: end_sys = [x^7-2,y^5-x^2]  # optional -- phc
sage: sol = phc.start_from(start_save, end_sys, R2)  # optional -- phc
sage: len(sol.solutions())  # optional -- phc
15

solution_dicts(get_failures=False)
Return a list of solutions in dictionary form: variable:value.

INPUT:

• self – for access to self_out_file_contents, the string of raw PHCpack output.

• get_failures (optional) – a boolean. The default (False) is to not process failed homotopies. These either lie on positive-dimensional components or at infinity.

OUTPUT:

• solution_dicts: a list of dictionaries. Each dictionary element is of the form variable:value, where the variable is an element of the input_ring, and the value is in ComplexField.

EXAMPLES:

sage: from sage.interfaces.phc import *
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: fs = [x^2-1,y^2-x,z^2-y]
sage: sol = phc.blackbox(fs,R)  # optional -- phc
sage: s_list = sol.solution_dicts()  # optional -- phc
sage: s_list.sort()  # optional -- phc
sage: s_list[0]  # optional -- phc
{y: 1.00000000000000, 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 (optional) – a boolean. The default (False) is to not process failed homotopies. These either lie on positive-dimensional components or at infinity.

OUTPUT:
• solutions: a list of lists of ComplexField-valued solutions.

EXAMPLES:
```
sage: from sage.interfaces.phc import *
sage: R2.<x1,x2> = PolynomialRing(QQ,2)
sage: test_sys = [x1^5-x1*x2^2-1, x2^5-x1*x2-1]
sage: sol = phc.blackbox(test_sys, R2)           # optional -- phc
sage: len(sol.solutions())                       # optional -- phc
2
```

variable_list()

Return the variables, as strings, in the order in which PHCpack has processed them.

EXAMPLES:
```
sage: from sage.interfaces.phc import *
sage: R2.<x1,x2> = PolynomialRing(QQ,2)
sage: test_sys = [x1^5-x1*x2^2-1, x2^5-x1*x2-1]
sage: sol = phc.blackbox(test_sys, R2)           # optional -- phc
sage: sol.variable_list()                       # optional -- phc
['x1', 'x2']
```

sage.interfaces.phc.get_classified_solution_dicts(output_file_contents, input_ring, get_failures=True)

Return a dictionary of lists of dictionaries of variable:value (key:value) pairs. Only used internally; see the classified_solution_dict function in the PHC_Object class definition for details.

INPUT:
• output_file_contents – phc solution output as a string
• input_ring – a PolynomialRing that variable names can be coerced into

OUTPUT:
• a dictionary of lists if dictionaries of solutions, classifies by type

EXAMPLES:
```
sage: from sage.interfaces.phc import *
sage: R2.<x1,x2> = PolynomialRing(QQ,2)
sage: test_sys = [(x1-2)^5-x2, (x2-1)^5-1]
sage: sol = phc.blackbox(test_sys, R2)           # optional -- phc
sage: sol_classes = get_classified_solution_dicts(sol.output_file_contents,R2)  # optional -- phc
sage: len(sol_classes['real'])                  # optional -- phc
1
```

sage.interfaces.phc.get_solution_dicts(output_file_contents, input_ring, get_failures=True)

Return a list of dictionaries of variable:value (key:value) pairs. Only used internally; see the solution_dict function in the PHC_Object class definition for details.

INPUT:
• output_file_contents – phc solution output as a string
• input_ring – a PolynomialRing that variable names can be coerced into

OUTPUT:
a list of dictionaries of solutions

EXAMPLES:

```python
sage: from sage.interfaces.phc import *
sage: R2.<x1,x2> = PolynomialRing(QQ,2)
sage: test_sys = [(x1-1)^5-x2, (x2-1)^5-1]
sage: sol = phc.blackbox(test_sys, R2)  # optional -- phc
sage: test = get_solution_dicts(sol.output_file_contents, R2)  # optional -- phc
sage: str(sum([q[x1].real() for q in test])[0:4])  # optional -- phc
'25.0'
```

`sage.interfaces.phc.get_variable_list(output_file_contents)`

Return the variables, as strings, in the order in which PHCpack has processed them.

EXAMPLES:

```python
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']
```
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 [AGHJLP2017].

```python
sage.interfaces.polymake.Polymake
alias of sage.interfaces.polymake.PolymakeExpect
```

```python
class sage.interfaces.polymake.PolymakeAbstract(seed=None)
    Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.interface.Interface

    Abstract interface to the polymake interpreter.

    This class should not be instantiated directly, but through its subclasses Polymake (Pexpect interface) or Poly-
    makeJuPyMake (JuPyMake interface).

    EXAMPLES:

```
We test the verbosity management with very early doctests because messages will not be repeated.

Testing the deprecated pexpect-based interface:

```
```
Testing the JuPyMake interface:

```
sage: isinstance(polymake_jupymake, PolymakeAbstract)
True
sage: p = polymake_jupymake.rand_sphere(4, 20, seed=5)  # optional - jupymake
sage: p
Random spherical polytope of dimension 4; seed=5...
```

```
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
```

**application** *(app)*

Change to a given polymake application.

**INPUT:**

- app, a string, one of “common”, “fulton”, “group”, “matroid”, “topaz”, “fan”, “graph”, “ideal”, “polytope”, “tropical”

**EXAMPLES:**

We expose a computation that uses both the ‘polytope’ and the ‘fan’ application of polymake. Let us start by defining a polytope $q$ in terms of inequalities. Polymake knows to compute the f- and h-vector and finds that the polytope is very ample:

```
sage: q = polymake.new_object("Polytope", INEQUALITIES=[[5,-4,0,1],[-3,0,-4,1],[-2,1,0,0],[-4,4,4,-1],[0,0,1,0],[8,0,0,-1],[1,0,-1,0],[3,-1,0,0]])  # optional - polymake
```

```
sage: q.H_VECTOR  # optional - polymake
1 5 5 1
```

```
sage: q.F_VECTOR  # optional - polymake
8 14 8
```

```
sage: q.VERY_AMPLE  # optional - polymake
true
```

In the application ‘fan’, polymake can now compute the normal fan of $q$ and its (primitive) rays:

```
sage: polymake.application('fan')  # optional - polymake
sage: g = q.normal_fan()  # optional - polymake
sage: g.RAYS  # optional - polymake
-1 0 1/4
0 -1 1/4
1 0 0
1 1 -1/4
0 1 0
0 0 -1
```
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):
  ... Not ImplementedError: 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 - polymake
\rightarrow 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 - polymake
'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 - polymake
cube of dimension 3
```

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

Displays polymake’s help on a given topic, as a string.

**INPUT:**
**topic**, a string

**pager**, optional bool, default True: When True, display help, otherwise return as a string.

**EXAMPLES:**

```
sage: print(polymake.help('Polytope', pager=False))          # optional - polymake
←polymake # 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 - polymake
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 - polymake
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: q = polymake.new_object("Polytope<Rational>", INEQUALITIES=[[4,-4,0,1],[-
-4,0,-4,1],[-2,1,0,0],[-4,4,4,-1],[0,0,1,0],[8,0,0,-1]]) # optional - polymake
sage: q.N_VERTICES                                          # optional - polymake
4
sage: q.BOUNDED                                             # optional - polymake
```

(continues on next page)
true

```sage
sage: q.VERTICES  # optional - polymake
1 2 0 4
1 3 0 8
1 2 1 8
1 3 1 8
sage: q.full_typename()  # optional - polymake
'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
sage: c = polymake('cube(3)')  # optional - polymake # indirect doctest
sage: d = polymake.cube(3)      # optional - polymake
```

Equality is, for “big” objects such as polytopes, comparison by identity:

```sage
sage: c == d  # optional - polymake
False
```

However, the list of vertices is equal:

```sage
sage: c.VERTICES == d.VERTICES  # optional - polymake
True
```

**version()**

Version of the polymake installation.

**EXAMPLES:**

```sage
sage: polymake.version()  # optional - polymake # random
'4...
```

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

Bases: `sage.interfaces.tab_completion.ExtraTabCompletion`, `sage.interfaces.interface.InterfaceElement`
Elements in the polymake interface.

EXAMPLES:

We support all “big” polymake types, Perl arrays of length different from one, and Perl scalars:

```
sage: p = polymake.rand_sphere(4, 20, seed=5) # optional - polymake
sage: p.typename() # optional - polymake
'Polytope'
sage: p # optional - polymake
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 - polymake
1450479926727001/2251799813685248
```

```
full_typename()
The name of the specialised type of this element.

EXAMPLES:

```
sage: c = polymake.cube(4) # optional - polymake
sage: c.full_typename() # optional - polymake
'Polytope<Rational>'
sage: c.VERTICES.full_typename() # optional - polymake
'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 - polymake
Normally, a property would be accessed as follows:
```
```
sage: p.F_VECTOR # optional - polymake
20 94 148 74
```
```
However, explicit access is possible as well:
```
```
sage: p.get_member('F VECTOR') # optional - polymake
20 94 148 74
```
```
In some cases, the explicit access works better:
```
```
sage: p.type # optional - polymake
Member function 'type' of Polymake::polytope::Polytope__Rational object
sage: p.get_member('type') # optional - polymake
Polytope<Rational>[SAGE...]
sage: p.get_member('type').get_member('name') # optional - polymake
Polytope
```

Note that in the last example calling the erroneously constructed member function `type` still works:

```
sage: p.type() # optional - polymake
Polytope<Rational>[
```

**get_member_function(attrname)**
Request a member function of this element.

**Note:** It is not checked whether a member function with the given name exists.

**EXAMPLES:**

```
sage: c = polymake.cube(2) # optional - polymake
sage: c.contains # optional - polymake
sage: V = polymake.new_object('Vector', [1,0,0]) # optional - polymake
sage: V # optional - polymake
1 0 0
sage: c.contains(V) # optional - polymake
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 - polymake
Member function 'foo' of Polymake::polytope::Polytope__Rational object
sage: c.get_member_function("foo")() # optional - polymake
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 - polymake
sage: c.known_properties() # optional - polymake
['AFFINE_HULL', 'BOUNDED',
```

(continues on next page)
A computation can change the list of known properties:

```
sage: c.F_VECTOR                           # optional - polymake
16 32 24 8
sage: c.known_properties()                  # optional - polymake
['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 - polymake
sage: c.qualified_typename()                 # optional - polymake
'polytope::Polytope<Rational>'
```

**typename()**

The name of the underlying base type of this element in polymake.

**EXAMPLES:**

```
sage: c = polymake.cube(4)                   # optional - polymake
sage: c.typename()                           # optional - polymake
'Polytope'
```

**typeof()**

Return the type of a polymake “big” object, and its underlying Perl type.

**Note:** This is mainly for internal use.

**EXAMPLES:**

```
sage: p = polymake.rand_sphere(3, 13, seed=12)  # optional - polymake
sage: p.typeof()                                 # optional - polymake
```

(continues on next page)
On "small" objects, it just returns empty strings:

\[
\text{sage: } p.\text{N\_VERTICES}.\text{typeof()}
\]
\[
\text{# optional - }\text{polymake}
\]
\[
\text{''} \quad \text{''}
\]

\[
\text{sage: } p.\text{list\_properties().typeof()}
\]
\[
\text{# optional - }\text{polymake}
\]
\[
\text{''} \quad \text{''}
\]

**exception** sage.interfaces.polymake.PolymakeError

Bases: RuntimeError

Raised if polymake yields an error message.

**class** sage.interfaces.polymake.PolymakeExpect

Bases: sage.interfaces.polymake.PolymakeAbstract, sage.interfaces.expect.Expect

Interface to the polymake interpreter using pexpect.

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.

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:**

\[
\text{sage: from sage.interfaces.polymake import polymake\_expect as polymake}
\]
\[
\text{sage: type(polymake)}
\]
\[
\text{<...sage.interfaces.polymake.PolymakeExpect...}
\]
\[
\text{sage: p = polymake.rand\_sphere(4, 20, seed=5)}
\]
\[
\text{# optional - polymake\_expect}
\]
\[
\text{sage: p}
\]
\[
\text{# optional - polymake\_expect}
\]
\[
\text{Random spherical polytope of dimension 4; seed=5...}
\]
\[
\text{sage: set\_verbose(3)}
\]
\[
\text{# optional - polymake\_expect #}
\]
\[
\text{sage: p.H\_VECTOR;}
\]
\[
\text{# optional - polymake\_expect #}
\]
\[
\text{random}
\]
\[
\text{used package ppl}
\]
\[
\text{The Parma Polyhedra Library ...}
\]
\[
\text{sage: p.H\_VECTOR}
\]
\[
\text{# optional - polymake\_expect}
\]
\[
1 16 40 16 1
\]
\[
\text{sage: set\_verbose(0)}
\]
\[
\text{# optional - polymake\_expect}
\]
\[
\text{sage: p.F\_VECTOR}
\]
\[
\text{# optional - polymake\_expect}
\]
\[
20 94 148 74
\]
\[
\text{sage: print(p.F\_VECTOR._sage\_doc_())}
\]
\[
\text{# optional - polymake\_expect #}
\]
\[
\text{random}
\]
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.:

\[
\text{
$v = \text{new \ Vector<Int>(1,2,3);}$
\]

or

\[
\text{
$v = \text{new \ Vector<Int>([1,2,3]);}$
\]

_eval_line_(line, allow_use_file=True, wait_for_prompt=True, restart_if_needed=True, **kwds)

Evaluate a command.

**INPUT:**

- line – string; a command to be evaluated
- allow_use_file – (default: True) bool; whether or not to use a file if the line is very long
- wait_for_prompt – (default: True) bool; whether or not to wait before polymake returns a prompt.
  If it is a string, it is considered as alternative prompt to be waited for
- restart_if_needed (default: True) bool; whether or not to restart polymake in case something goes wrong
- further optional arguments (e.g., timeout) that will be passed to pexpect.pty.spawn.spawn.
  expect(). Note that they are ignored if the line is too long and thus is evaluated via a file. So, if a timeout is defined, it should be accompanied by allow_use_file=False.

Different reaction types of polymake, including warnings, comments, errors, request for user interaction, and yielding a continuation prompt, are taken into account.

Usually, this method is indirectly called via eval().

**EXAMPLES:**

```python
sage: from sage.interfaces.polymake import polymake_expect as polymake
sage: p = polymake.cube(3) # optional - polymake_expect
...indirect doctest
```

Here we see that remarks printed by polymake are displayed if the verbosity is positive:

```python
sage: set_verbose(1)
sage: p.N_LATTICE_POINTS # optional - polymake_expect # 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)
```

If polymake raises an error, the polymake interface raises a PolymakeError:

```python
sage: polymake.eval('FOOBAR(3);') # optional - polymake_expect
Traceback (most recent call last):
```

(continues on next page)
If a command is incomplete, then polymake returns a continuation prompt. In that case, we raise an error:

```
sage: polymake.eval('print 3')  # optional - polymake_expect
Traceback (most recent call last):
...  
SyntaxError: Incomplete polymake command 'print 3'
sage: polymake.eval('print 3;')  # optional - polymake_expect
'3'
```

However, if the command contains line breaks but eventually is complete, no error is raised:

```
sage: print(polymake.eval('$tmp="abc";
print $tmp;'))  # optional - polymake_ 
˓→expect
abc
```

When requesting help, polymake sometimes expect the user to choose from a list. In that situation, we abort with a warning, and show the list from which the user can choose; we could demonstrate this using the `help()` method, but here we use an explicit code evaluation:

```
sage: print(polymake.eval('help "TRIANGULATION";'))  # optional - polymake_ 
˓→expect # 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 - polymake_expect
sage: polymake.eval('print { }-F_VECTOR;'.format(c.name()), timeout=1)  # not_ 
˓→tested # optional - polymake_expect
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 - polymake_expect
cube of dimension 15
sage: c.N_VERTICES  # optional - polymake_expect
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.

**application**(app)
Change to a given polymake application.

**INPUT:**
- app, a string, one of “common”, “fulton”, “group”, “matroid”, “topaz”, “fan”, “graph”, “ideal”, “polytope”, “tropical”

**EXAMPLES:**
We expose a computation that uses both the ‘polytope’ and the ‘fan’ application of polymake. Let us start by defining a polytope \( q \) in terms of inequalities. Polymake knows to compute the \( f \)- and \( h \)-vector and finds that the polytope is very ample:

```
sage: from sage.interfaces.polymake import polymake_expect as polymake
sage: q = polymake.new_object("Polytope", INEQUALITIES=[[5,-4,0,1],[-3,0,-4,1],
         →[-2,1,0,0],[-4,4,4,-1],[0,0,1,0],[8,0,0,-1],[1,0,-1,0],[3,-1,0,0]]) #
         →optional - polymake_expect
sage: q.H_VECTOR                    # optional - polymake_expect
1 5 5 1
sage: q.F_VECTOR                    # optional - polymake_expect
8 14 8
sage: q.VERY_AMPLE                  # optional - polymake_expect
true
```

In the application ‘fan’, polymake can now compute the normal fan of \( q \) and its (primitive) rays:

```
sage: polymake.application('fan')   # optional - polymake_expect
sage: g = q.normal_fan()            # optional - polymake_expect
sage: g.RAYS                        # optional - polymake_expect
-1 0 1/4
0 -1 1/4
1 0 0
1 1 -1/4
0 1 0
0 0 -1
0 -1 0
-1 0 0
sage: g.RAYS.primitive()            # optional - polymake_expect
-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.

**class** sage.interfaces.polymake.PolymakeFunctionElement(obj, name, memberfunction=False)
**Bases:** sage.interfaces.interface.InterfaceFunctionElement

A callable (function or member function) bound to a polymake element.
EXAMPLES:

```plaintext
sage: c = polymake.cube(2)  # optional - polymake
sage: V = polymake.new_object('Vector', [1,0,0])  # optional - polymake
sage: V
1 0 0  # optional - polymake
sage: c.contains  # optional - polymake
Member function 'contains' of Polymake::polytope::Polytope__Rational object
sage: c.contains(V)  # optional - polymake
true
```

class sage.interfaces.polymake.PolymakeJuPyMake(seed=None, verbose=False)

Bases: sage.interfaces.polymake.PolymakeAbstract

Interface to the polymake interpreter using JuPyMake.

In order to use this interface, you need to either install the optional polymake package for Sage, or install polymake system-wide on your computer; it is available from https://polymake.org. Also install the jupymake Python package.

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

EXAMPLES:

```plaintext
sage: from sage.interfaces.polymake import polymake_jupymake as polymake
sage: type(polymake)
<...sage.interfaces.polymake.PolymakeJuPyMake>
```

```plaintext
sage: p = polymake.rand_sphere(4, 20, seed=5)  # optional - jupymake
sage: p
Random spherical polytope of dimension 4; seed=5...
```

```plaintext
sage: set_verbose(3)
sage: p.H_VECTOR;  # optional - jupymake # random
used package ppl
The Parma Polyhedra Library ...
```

```plaintext
sage: p.H_VECTOR  # optional - jupymake
1 16 40 16 1
```

```plaintext
sage: set_verbose(0)
sage: p.F_VECTOR  # optional - jupymake
20 94 148 74
```

```plaintext
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]);
```
```
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.
```
eval(code, **kwd)
Evaluate a command.

INPUT:

- code -- a command (string) to be evaluated

Different reaction types of polymake, including warnings, comments, errors, request for user interaction, and yielding a continuation prompt, are taken into account.

EXAMPLES:

```
sage: from sage.interfaces.polymake import polymake_jupymake as polymake
sage: p = polymake.cube(3)  # optional - jupymake # indirect
˓→ doctest
```

Here we see that remarks printed by polymake are displayed if the verbosity is positive:

```
sage: set_verbose(1)
sage: p.N_LATTICE_POINTS  # optional - jupymake # random
```

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/

```
sage: set_verbose(0)
```

If polymake raises an error, the polymake interface raises a PolymakeError:

```
sage: polymake.eval('FOOBAR(3);')  # optional - jupymake
```

```
Traceback (most recent call last):
...
PolymakeError: Undefined subroutine &Polymake::User::FOOBAR called...
```

If a command is incomplete, then polymake returns a continuation prompt. In that case, we raise an error:
However, if the command contains line breaks but eventually is complete, no error is raised:

```
sage: print(polymake.eval('$tmp=“abc”;

print $tmp;'))  # optional - jupymake
abc
```

When requesting help, polymake sometimes expect the user to choose from a list. In that situation, we abort with a warning, and show the list from which the user can choose; we could demonstrate this using the `help()` method, but here we use an explicit code evaluation:

```
sage: print(polymake.eval('help "TRIANGULATION";'))  # optional - jupymake
˓→random
doctest:warning
...
UserWarning: Polymake expects user interaction. We abort and return the options that Polymake provides.
There are 5 help topics matching 'TRIANGULATION':
1: objects/Cone/properties/Triangulation and volume/TRIANGULATION
2: objects/Polytope/properties/Triangulation and volume/TRIANGULATION
3: objects/Visualization/Visual::PointConfiguration/methods/TRIANGULATION
4: objects/Visualization/Visual::Polytope/methods/TRIANGULATION
5: objects/PointConfiguration/properties/Triangulation and volume/TRIANGULATION
```

By default, we just wait until polymake returns a result. However, it is possible to explicitly set a timeout. The following usually does work in an interactive session and often in doc tests, too. However, sometimes it hangs, and therefore we remove it from the tests, for now:

```
sage: c = polymake.cube(15)  # optional - jupymake
sage: polymake.eval('print {}->FVECTOR;' .format(c.name()), timeout=1)  # nottested # optional - jupymake
Traceback (most recent call last):
...
RuntimeError: Polymake fails to respond timely
```

We verify that after the timeout, polymake is still able to give answers:

```
sage: c  # optional - jupymake
cube of dimension 15
sage: c.N_VERTICES  # optional - jupymake
32768
```

Note, however, that the recovery after a timeout is not perfect. It may happen that in some situation the interface collapses and thus polymake would automatically be restarted, thereby losing all data that have been computed before.

```
is_running()  
Return True if self is currently running.
```
sage.interfaces.polymake.polymake_console(command="")
Spawn a new polymake command-line session.

EXAMPLES:

```
sage: from sage.interfaces.polymake import polymake_console
sage: polymake_console()  # not tested
Welcome to polymake version ...
...
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
```

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
THIRTYSEVEN

PARALLEL INTERFACE TO THE SAGE INTERPRETER

This is an expect interface to emph{multiple} copy of the sage interpreter, which can all run simultaneous calculations. A PSage object does not work as well as the usual Sage object, but does have the great property that when you construct an object in a PSage you get back a prompt immediately. All objects constructed for that PSage print <<currently executing code>> until code execution completes, when they print as normal.

note{BUG– currently non-idle PSage subprocesses do not stop when sage exits. I would very much like to fix this but don’t know how.}

EXAMPLES:
We illustrate how to factor 3 integers in parallel. First start up 3 parallel Sage interfaces:

```
sage: v = [PSage() for _ in range(3)]
```

Next, request factorization of one random integer in each copy.

```
sage: w = [x('factor(2^%s-1)'% randint(250,310)) for x in v] # long time (5s on sage. →math, 2011)
```

Print the status:

```
sage: w  # long time, random output (depends on timing)
[3 * 11 * 31^2 * 311 * 11161 * 11471 * 73471 * 715827883 * 2147483647 * 4649919401 * →
  18158209813151 * 5947603221397891 * 29126056043168521,
  <<currently executing code>>,
  9623 * 68492481833 * →
  23579543011798993222850893929565870383844167873851502677311057483194673]
```

Note that at the point when we printed two of the factorizations had finished but a third one hadn’t. A few seconds later all three have finished:

```
sage: w  # long time, random output
[3 * 11 * 31^2 * 311 * 11161 * 11471 * 73471 * 715827883 * 2147483647 * 4649919401 * →
  18158209813151 * 5947603221397891 * 29126056043168521,
  23^2 * 47 * 89 * 178481 * 4103188409 * 199957736328435366769577 * →
  44667711762797798403039426178361,
  9623 * 68492481833 * →
  23579543011798993222850893929565870383844167873851502677311057483194673]
```

```
class sage.interfaces.psage.PSage(**kwds)
    Bases: sage.interfaces.sage0.Sage

eval(x, strip=True, **kwds)
    x – code strip – ignored
```
get(var)
    Get the value of the variable var.

is_locked()

set(var, value)
    Set the variable var to the given value.

class sage.interfaces.psage.PSageElement(parent, value, is_name=False, name=None)
    Bases: sage.interfaces.sage0.SageElement

    is_locked()
The basic function of QEPCAD is to construct cylindrical algebraic decompositions (CADs) of $\mathbb{R}^k$, 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 $\mathbb{R}^j$ 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 $\mathbb{R}^k$, and for each cell in $\mathbb{R}^j$ ($j > 1$), the projection of that cell into $\mathbb{R}^{j-1}$ is a cell of $\mathbb{R}^{j-1}$. (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 $\mathbb{R}^2$, find the cells of $\mathbb{R}^3$ 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 $\mathbb{R}^k$ 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.

```python
sage: var('a,b,c,d,x,y,z')
(a, b, c, d, x, y, z)
sage: qf = qepcad_formula
```

We start with a simple example. Consider an arbitrarily-selected ellipse:

```python
sage: ellipse = 3*x^2 + 2*x*y + y^2 - x + y - 7
```

What is the projection onto the $x$ axis of this ellipse? First we construct a formula asking this question.

```python
sage: ellipse = 3*x^2 + 2*x*y + y^2 - x + y - 7
```
Then we run qepcad to get the answer:

```python
sage: qepcad(F)  # optional - qepcad
8 x^2 - 8 x - 29 <= 0
```

How about the projection onto the $y$ axis?

```python
sage: qepcad(qf.exists(x, ellipse == 0))  # optional - qepcad
8 y^2 + 16 y - 85 <= 0
```

QEPCAD deals with more quantifiers than just ‘exists’, of course. Besides the standard ‘forall’, there are also ‘for infinitely many’, ‘for all but finitely many’, ‘for a connected subset’, and ‘for exactly k’. The `qepcad()` documentation has examples of all of these; here we will just give one example.

First we construct a circle:

```python
sage: circle = x^2 + y^2 - 3
```

For what values $k$ does a vertical line $x = k$ intersect the combined figure of the circle and ellipse exactly three times?

```python
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 \lor
-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 \(_\text{root}_k P(\bar{x}, y)\) refers to a particular zero of $P(\bar{x}, y)$ (viewed as a polynomial in $y$). If there are $n$ roots, then \(_\text{root}_1\) refers to the least root and \(_\text{root}_n\) refers to the greatest. Also, \(_\text{root}_{-n}\) refers to the least root and \(_\text{root}_{-1}\) 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.

```python
sage: qepcad(F, solution='geometric')  # not tested (random order)
x = _root_1 8 x^2 - 8 x - 29
\lor
8 x^4 - 26 x^2 - 4 x + 13 = 0
\lor
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:

```python
sage: F = qf.and_(ellipse < 0, circle < 0); F
[3 x^2 + 2 x y + y^2 - x + y - 7 < 0 \lor x^2 + y^2 - 3 < 0] (continues on next page)```
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.

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:

```python
sage: F = qf.exists(y, qf.and_(circle == 0, x + y > 0)); F
(E y)[x^2 + y^2 - 3 = 0 \∧ x + y > 0]
```

```python
sage: qepcad(F)         # not tested (random order)
x^2 - 3 <= 0 \∧ [ x > 0 \∨ 2 x^2 - 3 < 0 ]
```

Here, the formula \( x > 0 \) had to be introduced in order to get a solution formula; the original CAD of F did not include the polynomial \( x \). To avoid having QEPCAD do the extra work to come up with a solution formula, we can tell it to use the extended language; it is always possible to construct a solution formula in the extended language without introducing new polynomials.

```python
sage: qepcad(F, solution='extended')         # not tested (random order)
x^2 - 3 <= 0 \∧ [ x > _root_1 2 x^2 - 3
```

Up to this point, all the output we have seen has basically been in the form of strings; there is no support (yet) for parsing these outputs back into sage polynomials (partly because sage does not yet have support for symbolic conjunctions and disjunctions). The function \texttt{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, \texttt{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.

```python
sage: p = qepcad(ellipse == 0, solution='any-point'); p
{ '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 \( \mathbb{Q} \) (instead of a symbolic expression).

```python
sage: pellipse = QQ['x,y'](ellipse)
sage: pellipse(**p) == 0
True
```

For cell-points, let us look at points not on the ellipse.

```python
sage: pts = qepcad(ellipse != 0, solution='cell-points'); pts
{{'x': 4, 'y': 1},
 {'x': 2.468501968502953?, 'y': 9},
 {'x': 2.468501968502953?, 'y': -9},
 {'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.

```python
sage: [pellipse(**p) != 0 for p in pts]
[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.

```python
sage: F = qf.exactly_k(3, y, circle * ellipse == 0); F
(3 x^2 + 2 x y + y^2 - x + y - 7) (x^2 + y^2 - 3) = 0
```

```python
sage: pts = qepcad(F, solution='all-points'); pts
{{'x': 1.732050807568878?},
 {'x': 1.731054913462534?},
 {'x': 0.678911384208004?},
 {'x': -0.9417727377417167?},
 {'x': -1.468501968502953?},
 {'x': -1.468193559928821?}}
```

Since \( y \) is bound by the quantifier, the solutions only refer to \( x \).

We can substitute one of these solutions into the original equation:

```python
sage: pt = pts[0]
sage: pcombo = QQ['x,y'](circle * ellipse)
```

(continues on next page)
and verify that it does have three roots:

```
sage: intersections.roots()          # optional - qepcad
[(-4.403249005600958?, 1), (-0.06085260953679653?, 1), (0, 2)]
```

Let us check all six solutions.

```
sage: [len(pcombo(y=polygen(AA, 'y'), **p).roots()) for p in pts]  # optional - qepcad
[3, 3, 3, 3, 3, 3]
```

We said earlier that we can run QEPCAD either automatically or interactively. Now that we have discussed the automatic modes, let us turn to interactive uses.

If the `qepcad()` function is passed `interact=True`, then instead of returning a result, it returns an object of class `Qepcad` representing a running instance of QEPCAD that you can interact with. For example:

```
sage: qe = qepcad(qf.forall(x, x^2 + b*x + c > 0), interact=True); qe  # optional - qepcad
QEPCAD object in phase 'Before Normalization'
```

This object is a fairly thin wrapper over QEPCAD; most QEPCAD commands are available as methods on the `Qepcad` object. Given a `Qepcad` object `qe`, you can type `qe.[tab]` to see the available QEPCAD commands; to see the documentation for an individual QEPCAD command, for example `d_setting`, you can type `qe.d_setting?`. (In QEPCAD, this command is called `d-setting`. We systematically replace hyphens with underscores for this interface.)

The execution of QEPCAD is divided into four phases. Most commands are not available during all phases. We saw above that QEPCAD starts out in phase 'Before Normalization'. We see that the `d_cell` command is not available in this phase:

```
sage: qe.d_cell()          # optional - qepcad
Error GETCID: This command is not active here.
```

We will focus here on the fourth (and last) phase, 'Before Solution', because this interface has special support for some operations in this phase. Consult the QEPCAD documentation for information on the other phases.

We can tell QEPCAD to finish off the current phase and move to the next with its `go` command. (There is also the `step` command, which partially completes a phase for phases that have multiple steps, and the `finish` command, which runs QEPCAD to completion.)

```
sage: qe.go()  # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (x)'
sage: qe.go()  # optional - qepcad
QEPCAD object has moved to phase 'Before Choice'
sage: qe.go()  # optional - qepcad
QEPCAD object has moved to phase 'Before Solution'
```

Note that the `Qepcad` object returns the new phase whenever the phase changes, as a convenience for interactive use; except that when the new phase is 'EXITED', the solution formula printed by QEPCAD is returned instead.
Let us pick a nice, simple example, return to phase 4, and explore the resulting qe object.

We said before that QEPCAD creates ‘cylindrical algebraic decompositions’: since we have a bivariate polynomial, we get decompositions of $\mathbb{R}^0$, $\mathbb{R}^1$, and $\mathbb{R}^2$. 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 $\mathbb{R}^0$ is trivial (of course). The decomposition of $\mathbb{R}^1$ 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 $\mathbb{R}^0$.

These five cells give rise to five stacks in $\mathbb{R}^2$. 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 $\mathbb{R}^1$) and the interior of the circle has index (3,3).

We can view these cells with the QEPCAD command d_cell. For instance, let us look at the cell for the upper semicircle:

```
sage: qe.d_cell(3, 4)  # optional - qepcad
--------- Information about the cell (3,4) ---------
Level : 2
Dimension : 1
Number of children : 0
Truth value : T by trial evaluation.
Degrees after substitution : Not known yet or No polynomial.
Multiplicities : ((1,1))
Signs of Projection Factors
Level 1 : (-)
Level 2 : (0)
---------   Sample point       ---------
The sample point is in a PRIMITIVE representation.
alpha = the unique root of x^2 - 3 between 0 and 4
       = 1.7320508076-

Coordinate 1 = 0
            = 0.0000000000
Coordinate 2 = alpha
            = 1.7320508076-
```
We see that, the level of this cell is 2, meaning that it is part of the decomposition of $\mathbb{R}^2$. 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
QEPCAD cell (3, 4)   # optional - qepcad
```

A `QepcadCell` has accessor methods for the important state held within a cell. For instance:

```
sage: c.level()  # optional - qepcad
2
sage: c.index()  # optional - qepcad
(3, 4)
sage: qe.cell(3).number_of_children()  # optional - qepcad
5
sage: len(qe.cell(3))  # optional - qepcad
5
```

One particularly useful thing we can get from a cell is its sample point, as sage algebraic real numbers.

```
sage: c.sample_point()  # optional - qepcad
(0, 1.732050807568878?)
sage: c.sample_point_dict()  # optional - qepcad
{ 'x': 0, 'y': 1.732050807568878? }
```

We have seen that we can get cells using the `cell()` method. There are several QEPCAD commands that print lists of cells; we can also get cells using the `make_cells()` method, passing it the output of one of these commands.

```
sage: qe.make_cells(qe.d_true_cells())  # optional - qepcad
[QEPCAD cell (4, 2), QEPCAD cell (3, 4), QEPCAD cell (3, 2),
QEPCAD cell (2, 2)]
```

Also, the cells in the stack over a given cell can be accessed using array subscripting or iteration. (Remember that cells in a stack are numbered starting with one; we preserve this convention in the array-subscripting syntax.)

```
sage: c = qe.cell(3)  # optional - qepcad
sage: c[1]  # optional - qepcad
QEPCAD cell (3, 1)
sage: [c2 for c2 in c]  # optional - qepcad
[QEPCAD cell (3, 1), QEPCAD cell (3, 2), QEPCAD cell (3, 3),
QEPCAD cell (3, 4), QEPCAD cell (3, 5)]
```

We can do one more thing with a cell: we can set its truth value. Once the truth values of the cells have been set, we can get QEPCAD to produce a formula which is true in exactly the cells we have selected. This is useful if QEPCAD’s quantifier language is insufficient to express your problem.

For example, consider again our combined figure of the circle and the ellipse. Suppose you want to find all vertical lines that intersect the circle twice, and also intersect the ellipse twice. The vertical lines that intersect the circle twice can be found by simplifying:

```
sage: F = qf.exactly_k(2, y, circle == 0); F
(X2 y)[x^2 + y^2 - 3 = 0]
```
and the vertical lines that intersect the ellipse twice are expressed by:

```
sage: G = qf.exactly_k(2, y, ellipse == 0); G
(X2 y)[3 x^2 + 2 x y + y^2 - x + y - 7 = 0]
```

and the lines that intersect both figures would be:

```
sage: qf.and_(F, G)
```

Traceback (most recent call last):
...
ValueError: QEPCAD formulas must be in prenex (quantifiers outermost) form

...except that QEPCAD does not support formulas like this; in QEPCAD input, all logical connectives must be inside all quantifiers.

Instead, we can get QEPCAD to construct a CAD for our combined figure and set the truth values ourselves. (The exact formula we use doesn’t matter, since we’re going to replace the truth values in the cells; we just need to use a formula that uses both polynomials.)

```
sage: qe = qepcad(qf.and_(ellipse == 0, circle == 0), interact=True)  # optional - qepcad
sage: qe.go(); qe.go(); qe.go()  # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
```

Now we want to find all cells $c$ in the decomposition of $\mathbb{R}^4$ such that the stack over $c$ contains exactly two cells on the ellipse, and also contains exactly two cells on the circle.

Our input polynomials are ‘level-2 projection factors’, we see:

```
sage: qe.d_proj_factors()  # optional - qepcad
P_1,1 = fac(J_1,1) = fac(dis(A_2,1))
  = 8 x^2 - 8 x - 29
P_1,2 = fac(J_1,2) = fac(dis(A_2,2))
  = x^2 - 3
P_1,3 = fac(J_1,3) = fac(res(A_2,1|A_2,2))
  = 8 x^4 - 26 x^2 - 4 x + 13
A_2,1 = input
  = y^2 + 2 x y + y + 3 x^2 - x - 7
A_2,2 = input
  = y^2 + x^2 - 3
```

so we can test whether a cell is on the ellipse by checking that the sign of the corresponding projection factor is 0 in our cell. For instance, the cell (12,2) is on the ellipse:

```
sage: qe.cell(12,2).signs()[1][0]  # optional - qepcad
0
```

So we can update the truth values as desired like this:

```
sage: for c in qe.cell():  # optional - qepcad
    ....:    count_ellipse = 0
    ....:    count_circle = 0
    ....:    for c2 in c:
```

(continues on next page)
count_ellipse += (c2.signs()[1][0] == 0)

count_circle += (c2.signs()[1][1] == 0)

c.set_truth(count_ellipse == 2 and count_circle == 2)

and then we can get our desired solution formula. (The 'G' stands for 'geometric', and gives solutions using the same rules as solution='geometric' described above.)

sage: qe.solution_extension('G')
# not tested (random order)
8 x^2 - 8 x - 29 < 0
/

x^2 - 3 < 0

AUTHORS:

• Carl Witty (2008-03): initial version
• Thierry Monteil (2015-07) repackaging + noncommutative doctests.

class sage.interfaces.qepcad.Qepcad(formula, vars=None, logfile=None, verbose=False, memcells=None, server=None)
Bases: object

The wrapper for QEPCAD.

answer()
For a QEPCAD instance which is finished, return the simplified quantifier-free formula that it printed just before exiting.

EXAMPLES:

sage: qe = qepcad(x^3 - x == 0, interact=True)  # optional - qepcad
sage: qe.finish()  # not tested (random order)
x - 1 <= 0 /

x + 1 >= 0 /

[ x = 0 / 0 <= 0 /

x - 1 = 0 ]

sage: qe.answer()  # not tested (random order)
x - 1 <= 0 /

x + 1 >= 0 /

[ x = 0 / 0 <= 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
qe.assume(a != 0)  # optional - qepcad
sage: qe.finish()  # optional - qepcad
4 a c - b^2 <= 0
cell(*index)*
Given a cell index, returns a *QepcadCell* wrapper for that cell. Uses a cache for efficiency.

EXAMPLES:

```python
sage: qe = qepcad(x + 3 == 42, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'At the end of projection phase'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.cell(2) # optional - qepcad
QEPCAD cell (2)
sage: qe.cell(2) is qe.cell(2) # optional - qepcad
True
```

final_stats()
For a QEPCAD instance which is finished, return the statistics that it printed just before exiting.

EXAMPLES:

```python
sage: qe = qepcad(x == 0, interact=True) # optional - qepcad
sage: qe.finish() # optional - qepcad
x = 0
sage: qe.final_stats() # random, optional - qepcad
------------------------------------------------------------------------------
0 Garbage collections, 0 Cells and 0 Arrays reclaimed, in 0 milliseconds.
492840 Cells in AVAIL, 500000 Cells in SPACE.
System time: 8 milliseconds.
System time after the initialization: 4 milliseconds.
------------------------------------------------------------------------------
```

make_cells(text)
Given the result of some QEPCAD command that returns cells (such as d_cell(), d_witness_list(), etc.), return a list of cell objects.

EXAMPLES:

```python
sage: var('x,y')
(x, y)
sage: qe = qepcad(x^2 + y^2 == 1, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.make_cells(qe.d_false_cells()) # optional - qepcad
[QEPCAD cell (5, 1), QEPCAD cell (4, 3), QEPCAD cell (4, 1), QEPCAD cell (3, 5),
QEPCAD cell (3, 3), QEPCAD cell (3, 1), QEPCAD cell (2, 3), QEPCAD cell (2, 1),
QEPCAD cell (1, 1)]
```

phase()
Return the current phase of the QEPCAD program.

EXAMPLES:

```python
sage: qe = qepcad(x > 2/3, interact=True) # optional - qepcad
sage: qe.phase() # optional - qepcad
(continues on next page)
```
'Before Normalization'
sage: qe.go()  # optional - qepcad
QEPCAD object has moved to phase 'At the end of projection phase'
sage: qe.phase()  # optional - qepcad
'At the end of projection phase'
sage: qe.go()  # optional - qepcad
QEPCAD object has moved to phase 'Before Choice'
sage: qe.phase()  # optional - qepcad
'Before Choice'
sage: qe.go()  # optional - qepcad
QEPCAD object has moved to phase 'Before Solution'
sage: qe.phase()  # optional - qepcad
'Before Solution'
sage: qe.go(); qe.go(); qe.go()  # optional - qepcad
EXITED

set_truth_value(index, nv)
Given a cell index (or a cell) and an integer, set the truth value of the cell to that integer.
Valid integers are 0 (false), 1 (true), and 2 (undetermined).

EXAMPLES:

sage: qe = qepcad(x == 1, interact=True)  # optional - qepcad
sage: qe.go(); qe.go(); qe.go()  # optional - qepcad
QEPCAD object has moved to phase 'At the end of projection phase'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.set_truth_value(1, 1)  # optional - qepcad

solution_extension(kind)
The following documentation is modified from qepcad.help:

solution-extension x
Use an alternative solution formula construction method. The parameter x is allowed to be T, E, or G. If x is T, then a formula in the usual language of Tarski formulas is produced. If x is E, a formula in the language of Extended Tarski formulas is produced. If x is G, then a geometry-based formula is produced.

EXAMPLES:

sage: var('x,y')
(x, y)
sage: qf = qepcad_formula
sage: qe = qepcad(qf.and_(x^2 + y^2 - 3 == 0, x + y > 0), interact=True)  # optional - qepcad
sage: qe.go(); qe.go(); qe.go(); 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 \land y = _root_-1 y^2 _2 + x^2 - 3 ]
(continues on next page)
class sage.interfaces.qepcad.QepcadCell(parent, lines)
Bases: object
A wrapper for a QEPCAD cell.

index()
Give the index of a QEPCAD cell.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: qe = qepcad(x^2 + y^2 == 1, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.cell().index() # optional - qepcad
()  
sage: qe.cell(1).index() # optional - qepcad
(1,)  
sage: qe.cell(2, 2).index() # optional - qepcad
(2, 2)
```

level()
Return the level of a QEPCAD cell.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: qe = qepcad(x^2 + y^2 == 1, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
```
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.cell().level() # optional - qepcad
0
sage: qe.cell(1).level() # optional - qepcad
1
sage: qe.cell(2, 2).level() # optional - qepcad
2

number_of_children()  
Return the number of elements in the stack over a QEPCAD cell. (This is always an odd number, if the  
stack has been constructed.)

EXAMPLES:

sage: var('x,y')
(x, y)
sage: qe = qepcad(x^2 + y^2 == 1, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.cell().number_of_children() # optional - qepcad
5
sage: [c.number_of_children() for c in qe.cell()] # optional - qepcad
[1, 3, 5, 3, 1]

sample_point()  
Return the coordinates of a point in the cell, as a tuple of sage algebraic reals.

EXAMPLES:

sage: qe = qepcad(x^2 - x - 1 == 0, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'At the end of projection phase'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: v1 = qe.cell(2).sample_point()[0]; v1 # optional - qepcad
-0.618033988749895?
sage: v2 = qe.cell(4).sample_point()[0]; v2 # optional - qepcad
1.618033988749895?
sage: v1 + v2 == 1 # optional - qepcad
True

sample_point_dict()  
Return the coordinates of a point in the cell, as a dictionary mapping variable names (as strings) to sage  
algebraic reals.

EXAMPLES:

sage: qe = qepcad(x^2 - x - 1 == 0, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'At the end of projection phase'

(continues on next page)
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': 1.6180339887498957}

set_truth(v)
Set the truth value of this cell, as used by QEPCAD for solution formula construction.
The argument v should be either a boolean or None (which will set the truth value to 'undetermined').
EXAMPLES:
sage: var('x,y')
(x, y)
sage: qe = qepcad(x^2 + y^2 == 1, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.solution_extension('T') # optional - qepcad
y^2 + x^2 - 1 = 0
sage: qe.cell(3, 3).set_truth(True) # optional - qepcad
sage: qe.solution_extension('T') # optional - qepcad
y^2 + x^2 - 1 <= 0

signs()
Return the sign vector of a QEPCAD cell.
This is a list of lists. The outer list contains one element for each level of the cell; the inner list contains
one element for each projection factor at that level. These elements are either -1, 0, or 1.
EXAMPLES:
sage: var('x,y')
(x, y)
sage: qe = qepcad(x^2 + y^2 == 1, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: from sage.interfaces.qepcad import QepcadCell
sage: all_cells = flatten(qe.cell(), ltypes=QepcadCell, max_level=1) # optional - qepcad
sage: [(c, c.signs()[1][0]) for c in all_cells] # optional - qepcad
[(QEPCAD cell (1, 1), 1), (QEPCAD cell (2, 1), 1), (QEPCAD cell (2, 2), 0), (QEPCAD cell (2, 3), 1), (QEPCAD cell (3, 1), 1), (QEPCAD cell (3, 2), 0), (QEPCAD cell (3, 3), -1), (QEPCAD cell (3, 4), 0), (QEPCAD cell (3, 5), 1), (QEPCAD cell (4, 1), 1), (QEPCAD cell (4, 2), 0), (QEPCAD cell (4, 3), 1), (QEPCAD cell (5, 1), 1)]

class sage.interfaces.qepcad.QepcadFunction(parent, name)
    Bases: sage.interfaces.expect.ExpectFunction

A wrapper for a QEPCAD command.
class sage.interfaces.qepcad.Qepcad_expect

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

The low-level wrapper for QEPCAD.

sage.interfaces.qepcad.qepcad(formula, assume=None, interact=False, solution=None, vars=None, **kwargs)

Quantifier elimination and formula simplification using QEPCAD B.

If assume is specified, then the given formula is 'assumed', which is taken into account during final solution formula construction.

If interact=True is given, then a Qepcad object is returned which can be interacted with either at the command line or programmatically.

The type of solution returned can be adjusted with solution. The options are 'geometric', which tries to construct a solution formula with geometric meaning; 'extended', which gives a solution formula in an extended language that may be more efficient to construct; 'any-point', which returns any point where the formula is true; 'all-points', which returns a list of all points where the formula is true (or raises an exception if there are infinitely many); and 'cell-points', which returns one point in each cell where the formula is true.

All other keyword arguments are passed through to the Qepcad constructor.

For much more documentation and many more examples, see the module docstring for this module (type sage.interfaces.qepcad? to read this docstring from the sage command line).

The examples below require that the optional qepcad package is installed.

EXAMPLES:

sage: qf = qepcad_formula

sage: var('a,b,c,d,x,y,z,long_with_underscore_314159')
(a, b, c, d, x, y, z, long_with_underscore_314159)

sage: K.<q,r> = QQ[]

sage: qepcad('(E x)[a x + b > 0]', vars='(a,b,x)')    # not tested (random order)
a /= 0 \/ b > 0

sage: qepcad(a > b)                                    # optional - qepcad
b - a < 0

sage: qepcad(qf.exists(x, a*x^2 + b*x + c == 0))      # not tested (random order)
4 a c - b^2 <= 0 \/ \[ c = 0 \/ a /= 0 \/ 4 a c - b^2 < 0 \]

sage: qepcad(qf.exists(x, a*x^2 + b*x + c == 0), assume=(a != 0))   # optional - qepcad
4 a c - b^2 <= 0

For which values of a, b, c does ax^2 + bx + c have 2 real zeroes?

sage: exact2 = qepcad(qf.exactly_k(2, x, a*x^2 + b*x + c == 0)); exact2    # not tested (random order)
a /= 0 \/ 4 a c - b^2 < 0

one real zero?
```python
sage: exact1 = qepcad(qf.exactly_k(1, x, a*x^2 + b*x + c == 0)); exact1  # not tested (random order)
[ a > 0 \lor 4 a c - b^2 = 0 ] \lor [ a < 0 \lor 4 a c - b^2 = 0 ] \lor [ a = 0 \lor 4 a c - b^2 < 0 ]
```

No real zeroes?

```python
sage: exact0 = qepcad(qf.forall(x, a*x^2 + b*x + c != 0)); exact0  # not tested (random order)
4 a c - b^2 >= 0 \land c /= 0 \land [ b = 0 \lor 4 a c - b^2 > 0 ]
3^{75} real zeroes?
```

```python
sage: qepcad(qf.exactly_k(3^{75}, x, a*x^2 + b*x + c == 0))  # optional - qepcad
FALSE
```

We can check that the results don’t overlap:

```python
sage: qepcad(r'[[%s] \land [ %s ]]' % (exact0, exact1), vars='a,b,c')  # not tested (random order)
FALSE
sage: qepcad(r'[[%s] \land [ %s ]]' % (exact0, exact2), vars='a,b,c')  # not tested (random order)
FALSE
sage: qepcad(r'[[%s] \land [ %s ]]' % (exact1, exact2), vars='a,b,c')  # not tested (random order)
FALSE
```

...and that the union of the results is as expected:

```python
sage: qepcad(r'[[%s] \lor [%s] \lor [%s]]' % (exact0, exact1, exact2), vars=(a,b,c))  # not tested (random order)
b /= 0 \lor a /= 0 \lor 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.

```python
sage: qepcad(qf.infinitely_many(x, a*x^2 + b*x + c == 0))  # not tested (random order)
a = 0 \land b = 0 \land c = 0
```

The polynomial is nonzero almost everywhere iff it is not identically zero.

```python
sage: qepcad(qf.all_but_finitely_many(x, a*x^2 + b*x + c != 0))  # not tested (random order)
b /= 0 \land a /= 0 \land c /= 0
```

The non-zeroes are continuous iff there are no zeroes or if the polynomial is zero.

```python
sage: qepcad(qf.connected_subset(x, a*x^2 + b*x + c != 0))  # not tested (random order)
4 a c - b^2 >= 0 \land [ a = 0 \land 4 a c - b^2 > 0 ]
```

The zeroes are continuous iff there are no or one zeroes, or if the polynomial is zero:
Since polynomials are continuous and \( y > 0 \) is an open set, they are positive infinitely often if and only if they are positive at least once.

```sage
sage: qepcad(qf.infinitely_many(x, a*x^2 + b*x + c > 0))  # not tested
→(random order)
c > 0 \lor a > 0 \lor 4a c - b^2 < 0
```

However, since \( y \geq 0 \) is not open, the equivalence does not hold if you replace ‘positive’ with ‘nonnegative’. (We assume \( a \neq 0 \) to get simpler formulas.)

```sage
sage: qepcad(qf.infinitely_many(x, a*x^2 + b*x + c >= 0), assume=(a != 0))  # not tested
→tested (random order)
a > 0 \lor 4a c - b^2 < 0
```

sage.interfaces.qepcad.qepcad.banner()

Return the QEPCAD startup banner.

EXAMPLES:

```sage
sage: from sage.interfaces.qepcad import qepcad_banner
sage: qepcad_banner()  # optional - qepcad
=======================================================
Quantifier Elimination in Elementary Algebra and Geometry by Partial Cylindrical Algebraic Decomposition ...
by Hoon Hong (hhong@math.ncsu.edu) With contributions by: Christopher W. Brown, George E. Collins, Mark J. Encarnacion, Jeremy R. Johnson Werner Krandick, Richard Liska, Scott McCallum, Nicolas Robidoux, and Stanly Steinberg
=======================================================
```

sage.interfaces.qepcad.qepcad_console(memcells=None)

Run QEPCAD directly. To exit early, press Control-C.

EXAMPLES:
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_formulal, a symbolic equality or inequality, or a polynomial \( p \) (meaning \( p = 0 \)).

EXAMPLES:

```
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.forall(a, a^2 + b > b^2 + a)
(A a)[a^2 + b > b^2 + a]
sage: qf.forall((a, b), a^2 + b^2 > 0)
(A a)(A b)[a^2 + b^2 > 0]
sage: qf.A(b, b^2 != a)
(A b)[b^2 /= a]
```

C(v, formula, allow_multi=False)
Given a variable and a formula, returns a new formula which is true iff the set of values for the variable at which the original formula was true is connected (including cases where this set is empty or is a single point).

This method is available both as connected_subset() and C() (the QEPCAD name for this quantifier).

The input formula may be a qformula as returned by the methods of qepcad_formulal, a symbolic equality or inequality, or a polynomial \( p \) (meaning \( p = 0 \)).

EXAMPLES:

```
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.connected_subset(a, a^2 + b > b^2 + a)
(C a)[a^2 + b > b^2 + a]
sage: qf.C(b, b^2 != a)
(C b)[b^2 /= a]
```

E(v, formula)
Given a variable (or list of variables) and a formula, returns the existential quantification of the formula over the variables.

This method is available both as exists() and E() (the QEPCAD name for this quantifier).

The input formula may be a qformula as returned by the methods of qepcad_formulal, a symbolic equality or inequality, or a polynomial \( p \) (meaning \( p = 0 \)).
EXAMPLES:

```python
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.exists(a, a^2 + b > b^2 + a)
(E a)[a^2 + b > b^2 + a]
sage: qf.exists((a, b), a^2 + b^2 < 0)
(E a)(E b)[a^2 + b^2 < 0]
sage: qf.E(b, b^2 == a)
(E b)[b^2 = a]
```

\(F(v, \text{formula})\)

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 \texttt{infinitely_many()} and \(F()\) (the QEPCAD name for this quantifier).

The input formula may be a \texttt{qformula} as returned by the methods of \texttt{qepcad_formula}, a symbolic equality or inequality, or a polynomial \(p\) (meaning \(p = 0\)).

EXAMPLES:

```python
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.infinitely_many(a, a^2 + b > b^2 + a)
(F a)[a^2 + b > b^2 + a]
sage: qf.F(b, b^2 != a)
(F b)[b^2 /= a]
```

\(G(v, \text{formula})\)

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 \texttt{all_but_finitely_many()} and \(G()\) (the QEPCAD name for this quantifier).

The input formula may be a \texttt{qformula} as returned by the methods of \texttt{qepcad_formula}, a symbolic equality or inequality, or a polynomial \(p\) (meaning \(p = 0\)).

EXAMPLES:

```python
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.all_but_finitely_many(a, a^2 + b > b^2 + a)
(G a)[a^2 + b > b^2 + a]
sage: qf.G(b, b^2 != a)
(G b)[b^2 /= a]
```

\(X(k, v, \text{formula}, allow_multi=False)\)

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 \texttt{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 \texttt{forall()} and \texttt{not()}.)
The input formula may be a \textit{qformula} as returned by the methods of \texttt{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]
```

\textbf{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 \textit{all\_but\_finitely\_many()} and \textit{G()} (the QEPCAD name for this quantifier).

The input formula may be a \textit{qformula} as returned by the methods of \texttt{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]
```

\textbf{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 \textit{qformula} as returned by the methods of \texttt{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]]
```

\textbf{atomic}(lhs, op=’=’, rhs=0)

Construct a QEPCAD formula from the given inputs.

**INPUT:**

- \texttt{lhs} – a polynomial, or a symbolic equality or inequality
- \texttt{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:

```python
sage: qf = qepcad_formula
sage: var('a,b,c')
(a, b, c)
sage: K.<x,y> = QQ[]
sage: def test_qf(qf):
    ....:     return qf, qf.vars
sage: test_qf(qf.atomic(a^2 + 17))
(a^2 + 17 = 0, frozenset({'a'}))
sage: test_qf(qf.atomic(a*b*c <= c^3))
(a b c <= c^3, frozenset({'a', 'b', 'c'}))
sage: test_qf(qf.atomic(x+y^2, '!=', a+b))
(y^2 + x /= a + b, frozenset({'a', 'b', 'x', 'y'}))
sage: test_qf(qf.atomic(x, operator.lt))
(x < 0, frozenset({'x'}))
```

connected_subset(v, formula, allow_multi=False)

Given a variable and a formula, returns a new formula which is true iff the set of values for the variable at which the original formula was true is connected (including cases where this set is empty or is a single point).

This method is available both as connected_subset() and C() (the QEPCAD name for this quantifier).

The input formula may be a qformula as returned by the methods of qepcad_formula, a symbolic equality or inequality, or a polynomial p (meaning p = 0).

EXAMPLES:

```python
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.connected_subset(a, a^2 + b > b^2 + a)
(C a)[a^2 + b > b^2 + a]
sage: qf.C(b, b^2 != a)
(C b)[b^2 /= a]
```

exactly_k(k, v, formula, allow_multi=False)

Given a nonnegative integer k, a variable, and a formula, returns a new formula which is true iff the original formula is true for exactly k values of the variable.

This method is available both as exactly_k() and X() (the QEPCAD name for this quantifier).

(Note that QEPCAD does not support k = 0 with this syntax, so if k = 0 is requested we implement it with forall() and not_.() )

The input formula may be a qformula as returned by the methods of qepcad_formula, a symbolic equality or inequality, or a polynomial p (meaning p = 0).

EXAMPLES:
```
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.exactly_k(1, x, x^2 + a*x + b == 0)
(X1 x)[a x + x^2 + b = 0]
sage: qf.exactly_k(0, b, a*b == 1)
(A b)[-a b = 1]
```

**exists**\( (v, \text{formula}) \)
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 \texttt{exists()} and \texttt{E()} (the QEPCAD name for this quantifier).

The input formula may be a \texttt{qformula} as returned by the methods of \texttt{qepcad_formula}, a symbolic equality or inequality, or a polynomial \( p \) (meaning \( p = 0 \)).

**EXAMPLES:**
```
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.exists(a, a^2 + b > b^2 + a)
(E a)[a^2 + b > b^2 + a]
sage: qf.exists((a, b), a^2 + b^2 < 0)
(E a)(E b)[a^2 + b^2 < 0]
sage: qf.E(b, b^2 == a)
(E b)[b^2 = a]
```

**forall**\( (v, \text{formula}) \)
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 \texttt{forall()} and \texttt{A()} (the QEPCAD name for this quantifier).

The input formula may be a \texttt{qformula} as returned by the methods of \texttt{qepcad_formula}, a symbolic equality or inequality, or a polynomial \( p \) (meaning \( p = 0 \)).

**EXAMPLES:**
```
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.forall(a, a^2 + b > b^2 + a)
(A a)[a^2 + b > b^2 + a]
sage: qf.forall((a, b), a^2 + b^2 > 0)
(A a)(A b)[a^2 + b^2 > 0]
sage: qf.A(b, b^2 != a)
(A b)[b^2 /= a]
```

**formula**\( (\text{formula}) \)
Constructs a QEPCAD formula from the given input.

**INPUT:**
- \texttt{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 \land a /= 0]
```

**iff**\((f_1,f_2)\)

Return the equivalence of its input formulas (that is, given formulas \( P \) and \( Q \), returns ‘\( P \text{ iff } Q \)’).

The input formulas may be a \texttt{qformula} as returned by the methods of \texttt{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 \iff b = 0]
sage: qf.iff(a^2 < b, b^2 < a)
[a^2 < b \iff b^2 < a]
```

**implies**\((f_1,f_2)\)

Return the implication of its input formulas (that is, given formulas \( P \) and \( Q \), returns ‘\( P \text{ implies } Q \)’).

The input formulas may be a \texttt{qformula} as returned by the methods of \texttt{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 \implies b = 0]
sage: qf.implies(a^2 < b, b^2 < a)
[a^2 < b \implies b^2 < a]
```

**infinitely_many**\((v,\text{formula})\)

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 \texttt{infinitely_many()} and \texttt{F()} (the QEPCAD name for this quantifier).

The input formula may be a \texttt{qformula} as returned by the methods of \texttt{qepcad_formula}, a symbolic equality or inequality, or a polynomial \( p \) (meaning \( p = 0 \)).

**EXAMPLES:**

```
sage: var('a,b')
(a, b)
```

(continues on next page)
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
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.not_(a > b)
[-a > b]
sage: qf.not_(a^2 + b^2)
[-a^2 + b^2 = 0]
sage: qf.not_(qf.and_(a > 0, b < 0))
[~[a > 0 /\ b < 0]]
```

or_(*formulas)
Return the disjunction of its input formulas.
(This method would be named ‘or’ if that were not a Python keyword.)
Each input formula may be a qformula as returned by the methods of qepcad_formula, a symbolic equality or inequality, or a polynomial \( p \) (meaning \( p = 0 \)).

EXAMPLES:

```sage
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]]
```
sage.interfaces.qepcad.qepcad_version()
    Return a string containing the current QEPCAD version number.

    EXAMPLES:

    sage: qepcad_version()
    # random, optional - qepcad
    'Version B 1.69, 16 Mar 2012'

class sage.interfaces.qepcad.qformula(formula, vars, qvars=[])
    Bases: object

    A qformula holds a string describing a formula in QEPCAD’s syntax, and a set of variables used.
INTERFACE TO BILL HART’S QUADRATIC SIEVE

sage.interfaces.qsieve.data_to_list(out, n, time)
Convert output of Hart’s sieve and n to a list and time.

INPUT:
- out – snapshot of text output of Hart’s QuadraticSieve program
- n – the integer being factored

OUTPUT:
- list – proper factors found so far
- str – time information

sage.interfaces.qsieve.qsieve(n, block=True, time=False, verbose=False)
Run Hart’s quadratic sieve and return the distinct proper factors of the integer n that it finds.

CONDITIONS:
The conditions for the quadratic sieve to work are as follows:

- No small factors
- Not a perfect power
- Not prime

If any of these fails, the sieve will also.

INPUT:
- n – an integer with at least 40 digits
- block – (default: True) if True, you must wait until the sieve computation is complete before using Sage further. If False, Sage will run while the sieve computation runs in parallel. If q is the returned object, use q.quit() to terminate a running factorization.
- time – (default: False) if True, time the command using the UNIX “time” command (which you might have to install).
- verbose – (default: False) if True, print out verbose logging information about what happened during the Sieve run (for non-blocking Sieve, verbose information is always available via the log() method.)

OUTPUT:
- list – a list of the distinct proper factors of n found
- str – the time in cpu seconds that the computation took, as given by the command line time command. (If time is False, this is always an empty string.)

EXAMPLES:
sage: k = 19; n = next_prime(10^k)*next_prime(10^(k+1))
sage: factor(n)  # (currently) uses PARI
10000000000000000051 * 100000000000000000039
sage: v, t = qsieve(n, time=True)  # uses qsieve; optional - time
sage: v  # optional - time
[10000000000000000051, 100000000000000000039]
sage: t  # random; optional - time
'0.36 real 0.19 user 0.00 sys'

sage.interfaces.qsieve.qsieve_block(n, time, verbose=False)
Compute the factorization of n using Hart's quadratic Sieve blocking until complete.

class sage.interfaces.qsieve.qsieve_nonblock(n, time)
Bases: object
A non-blocking version of Hart's quadratic sieve.
The sieve starts running when you create the object, but you can still use Sage in parallel.

EXAMPLES:

sage: k = 19; n = next_prime(10^k)*next_prime(10^(k+1))
sage: q = qsieve(n, block=False, time=True)  # optional - time
sage: q  # random output; optional - time
Proper factors so far: []
sage: q  # random output; optional - time
([[10000000000000000051, 100000000000000000039], '0.21'])
sage: q.list()  # random output; optional - time
[10000000000000000051, 100000000000000000039]
sage: q.time()  # random output; optional - time
'0.21'

sage: q = qsieve(next_prime(10^20)*next_prime(10^21), block=False)
sage: q  # random output
Proper factors so far: [100000000000000000039, 1000000000000000000117]
sage: q  # random output
[100000000000000000039, 1000000000000000000117]

cputime()
Return the time in seconds (as a string) that it took to factor n, or return ‘?’ if the factorization has not completed or the time is unknown.
done()
Return True if the sieve process has completed.
list()
Return a list of the factors found so far, as Sage integers.
log()
Return all output of running the sieve so far.
n()
Return the integer that is being factored.
pid()
Return the PIN id of the QuadraticSieve process (actually of the time process that spawns the sieve process).
quit()
Terminate the QuadraticSieve process, in case you want to give up on computing this factorization.
EXAMPLES:

```
sage: n = next_prime(2^310)*next_prime(2^300)
sage: qs = qsieve(n, block=False)
sage: qs
Proper factors so far: []
sage: qs.quit()
sage: qs
Factorization was terminated early.
```

`time()`  
Return the time in seconds (as a string) that it took to factor \( n \), or return ‘?’ if the factorization has not completed or the time is unknown.
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 described in the Notebook manual.

The %R and %%R interface creating an R line or an R cell in the Jupyter notebook are briefly described at the end of this page. This documentation will be expanded and placed in the Jupyter notebook manual when this manual exists.

The following examples try to follow “An Introduction to R” which can be found at http://cran.r-project.org/doc/manuals/R-intro.html.

EXAMPLES:

Simple manipulations; numbers and vectors

The simplest data structure in R is the numeric vector which consists of an ordered collection of numbers. To create a vector named \( x \) using the R interface in Sage, you pass the R interpreter object a list or tuple of numbers:

```
sage: x = r([10.4,5.6,3.1,6.4,21.7]); x
[1] 10.4 5.6 3.1 6.4 21.7
```

You can invert elements of a vector \( x \) in R by using the invert operator or by doing \( 1/x \):

```
sage: ~x # optional - rpy2
[1] 0.09615385 0.17857143 0.32258065 0.15625000 0.04608295
sage: 1/x # optional - rpy2
[1] 0.09615385 0.17857143 0.32258065 0.15625000 0.04608295
```

The following assignment creates a vector \( y \) with 11 entries which consists of two copies of \( x \) with a 0 in between:

```
sage: y = r([x,0,x]); y # optional - rpy2
[1] 10.4 5.6 3.1 6.4 21.7 0.0 10.4 5.6 3.1 6.4 21.7
```

Vector Arithmetic

The following command generates a new vector \( v \) of length 11 constructed by adding together (element by element) \( 2x \) repeated 2.2 times, \( y \) repeated just once, and 1 repeated 11 times:

```
sage: v = 2*x+y+1; v # optional - rpy2
[1] 32.2 17.8 10.3 20.2 66.1 21.8 22.6 12.8 16.9 50.8 43.5
```

One can compute the sum of the elements of an R vector in the following two ways:

```
sage: sum(x) # optional - rpy2
[1] 47.2
```

(continues on next page)
sage: x.sum()  # optional - rpy2
[1] 47.2

One can calculate the sample variance of a list of numbers:

sage: ((x-x.mean())^2/(x.length()-1)).sum()  # optional - rpy2
[1] 53.853
sage: x.var()  # optional - rpy2
[1] 53.853

sage: x.sort()  # optional - rpy2
[1] 3.1 5.6 6.4 10.4 21.7
sage: x.min()  # optional - rpy2
[1] 3.1
sage: x.max()  # optional - rpy2
sage: x  # optional - rpy2
[1] 10.4 5.6 3.1 6.4 21.7

sage: r(-17).sqrt()  # optional - rpy2
[1] NaN
sage: r('-17+0i').sqrt()  # optional - rpy2
[1] 0+4.123106i

Generating an arithmetic sequence:

sage: r('1:10')  # optional - rpy2
[1] 1 2 3 4 5 6 7 8 9 10

Because `from` is a keyword in Python, it can’t be used as a keyword argument. Instead, `from_` can be passed, and R will recognize it as the correct thing:

sage: r.seq(length=10, from_=-1, by=.2)  # optional - rpy2
[1] -1.0 -0.8 -0.6 -0.4 -0.2 0.0 0.2 0.4 0.6 0.8
sage: x = r([10.4,5.6,3.1,6.4,21.7])  # optional - rpy2
sage: x.rep(2)  # optional - rpy2
[1] 10.4 5.6 3.1 6.4 21.7 10.4 5.6 3.1 6.4 21.7
sage: x.rep(times=2)  # optional - rpy2
[1] 10.4 5.6 3.1 6.4 21.7 10.4 5.6 3.1 6.4 21.7
sage: x.rep(each=2)  # optional - rpy2
[1] 10.4 10.4 5.6 5.6 3.1 3.1 6.4 6.4 21.7 21.7

Missing Values:

sage: na = r('NA')  # optional - rpy2
sage: z = r([1,2,3,na])  # optional - rpy2
sage: z  # optional - rpy2
[1] 1 2 3 NA
sage: ind = r.is_na(z)  # optional - rpy2
sage: ind  # optional - rpy2
[1] FALSE FALSE FALSE TRUE
sage: zero = r(0)  # optional - rpy2

(continues on next page)
sage: zero / zero  # optional - rpy2
[1] NaN
sage: inf = r('Inf')  # optional - rpy2
sage: inf-inf  # optional - rpy2
[1] NaN
sage: r.is_na(inf)  # optional - rpy2
[1] FALSE
sage: r.is_na(inf-inf)  # optional - rpy2
[1] TRUE
sage: r.is_na(zero/zero)  # optional - rpy2
[1] TRUE
sage: r.is_na(na)  # optional - rpy2
[1] TRUE
sage: r.is_nan(inf-inf)  # optional - rpy2
[1] TRUE
sage: r.is_nan(zero/zero)  # optional - rpy2
[1] TRUE
sage: r.is_nan(na)  # optional - rpy2
[1] FALSE

Character Vectors:

sage: labs = r.paste('c("X","Y")', '1:10', sep='""'); labs  # optional - rpy2
[1] "X1" "Y2" "X3" "Y4" "X5" "Y6" "X7" "Y8" "X9" "Y10"

Index vectors; selecting and modifying subsets of a data set:

sage: na = r('NA')  # optional - rpy2
sage: x = r([10.4,5.6,3.1,6.4,21.7,na]); x  # optional - rpy2
[1] 10.4 5.6 3.1 6.4 21.7 NA
sage: x[!is.na(self)]  # optional - rpy2
[1] 10.4 5.6 3.1 6.4 21.7
sage: x = r([10.4,-2,3.1,-0.5,21.7,na]); x  # optional - rpy2
[1] 10.4 -2.0 3.1 -0.5 21.7 NA
sage: (x+1)[!(is.na(self)) & self>0]  # optional - rpy2
[1] 11.4 4.1 0.5 22.7

Distributions:

sage: r.options(width="60")  # optional - rpy2
$width
[1] 80
sage: rr = r.dnorm(r.seq(-3,3,0.1))  # optional - rpy2
sage: rr  # optional - rpy2
[1] 0.004431848 0.005952532 0.007915452 0.010420935
[5] 0.013582969 0.017528300 0.022394530 0.028327038
[9] 0.035474593 0.043983596 0.053990967 0.065615815

(continues on next page)
Convert R Data Structures to Python/Sage:

```python
sage: rr = r.dnorm(r.seq(-3,3,0.1)) # optional - rpy2
sage: sum(rr._sage_()) # optional - rpy2
9.9772125168981...
```

Or you get a dictionary to be able to access all the information:

```python
sage: rs = r.summary(r.c(1,4,3,4,3,2,5,1)) # optional - rpy2
sage: rs # optional - rpy2

Min. 1st Qu. Median Mean 3rd Qu. Max.
1.000 1.750 3.000 2.875 4.000 5.000

sage: d = rs._sage_() # optional - rpy2
sage: d['DATA'] # optional - rpy2
[1, 1.75, 3, 2.875, 4, 5]

sage: d['_Names'] # optional - rpy2
['Min.', '1st Qu.', 'Median', 'Mean', '3rd Qu.', 'Max.]

sage: d['_r_class'] # optional - rpy2
['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 by other r... functions.

• Data exchanged via the -i and -o options have a format DIFFERENT from the format used by the r... functions (RPy2 mostly uses arrays, and bugs the user to use the pandas Python package).

• R graphics are (beautifully) displayed in output cells, but are not directly importable. You have to save them as .png, .pdf or .svg files and import them in Sage for further use.

In its current incarnation, this interface is mostly useful to statisticians needing Sage for a few symbolic computations but mostly using R for applied work.

AUTHORS:

• Mike Hansen (2007-11-01)
• William Stein (2008-04-19)
• Harald Schilly (2008-03-20)
• Mike Hansen (2008-04-19)
• Emmanuel Charpentier (2015-12-12, RPy2 interface)

class sage.interfaces.r.HelpExpression
Bases: str

Used to improve printing of output of r.help.

class sage.interfaces.r.R(maxread=None, logfile=None, init_list_length=1024, seed=None)
Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.interface.Interface

An interface to the R interpreter.

R is a comprehensive collection of methods for statistics, modelling, bioinformatics, data analysis and much more. For more details, see http://www.r-project.org/about.html

Resources:

• http://r-project.org/ provides more information about R.

• http://rseek.org/ R’s own search engine.

EXAMPLES:

```
sage: r.summary(r.c(1,2,3,111,2,3,2,3,2,5,4))  # optional - rpy2

<table>
<thead>
<tr>
<th></th>
<th>Min. 1st Qu.</th>
<th>Median</th>
<th>Mean</th>
<th>3rd Qu.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.00</td>
<td>2.00</td>
<td>3.00</td>
<td>12.55</td>
<td>3.50</td>
</tr>
<tr>
<td></td>
<td>111.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

available_packages()

Returns a list of all available R package names.

This list is not necessarily sorted.

OUTPUT: list of strings

Note: This requires an internet connection. The CRAN server is that is checked is defined at the top of sage/interfaces/r.py.

EXAMPLES:
**sage:** `ap = r.available_packages()`  # optional - internet  # optional - rpy2
**sage:** `len(ap) > 20`  # optional - internet  # optional - rpy2
True

**call**(function_name, *args, **kwds)
This is an alias for function_call().

EXAMPLES:

**sage:** `r.call('length', [1,2,3])`  # optional - rpy2
[1] 3

**chdir**(dir)
Changes the working directory to dir

INPUT:

• dir – the directory to change to.

EXAMPLES:

**sage:** `import tempfile`  # optional - rpy2
**sage:** `tmpdir = tempfile.mkdtemp()`  # optional - rpy2
**sage:** `r.chdir(tmpdir)`  # optional - rpy2

Check that tmpdir and r.getwd() refer to the same directory. We need to use realpath() in case $TMPDIR (by default /tmp) is a symbolic link (see trac ticket #10264).

**sage:** `os.path.realpath(tmpdir) == sageobj(r.getwd())`  # known bug (trac #9970)
˓→ True

**completions**(s)
Return all commands names that complete the command starting with the string s. This is like typing s[Ctrl-T] in the R interpreter.

INPUT:

• s – string

OUTPUT: list – a list of strings

EXAMPLES:

**sage:** `dummy = r._tab_completion(use_disk_cache=False)`  # clean doctest #
˓→ optional - rpy2
**sage:** `'testInheritedMethods' in r.completions('tes')`  # optional - rpy2
True

**console**()
Runs the R console as a separate new R process.

EXAMPLES:

**sage:** `r.console()`  # not tested  # optional - rpy2
R version 2.6.1 (2007-11-26)
Copyright (C) 2007 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
...
**convert_r_list(l)**
Converting an R list to a Python list.

**EXAMPLES:**

```sage
s = 'c(".GlobalEnv", "package:stats", "package:graphics",
    "Autoloads", \n"package:base")'  # optional - rpy2
sage: r.convert_r_list(s)  # optional - rpy2
['.GlobalEnv',
 'package:stats',
 'package:graphics',
 'package:grDevices',
 'package:utils',
 'package:datasets',
 'package:methods',
 'Autoloads',
 'package:base']
```

**eval(code, *args, **kwds)**
Evaluates a command inside the R interpreter and returns the output as a string.

**EXAMPLES:**

```sage
sage: r.eval('1+1')  # optional - rpy2
[1] 2
```

**function_call(function, args=None, kwds=None)**
Return the result of calling an R function, with given args and keyword args.

**OUTPUT:** RElement – an object in R

**EXAMPLES:**

```sage
sage: r.function_call('length', args=[[1,2,3]])  # optional - rpy2
[1] 3
```

**get(var)**
Returns the string representation of the variable var.

**INPUT:**

- var – a string

**OUTPUT:** string

**EXAMPLES:**

```sage
sage: r.set('a', 2)  # optional - rpy2
sage: r.get('a')  # optional - rpy2
[1] 2
```

**help(command)**
Returns help string for a given command.

**INPUT:** command – a string

**OUTPUT:** HelpExpression – a subclass of string whose __repr__ method is __str__, so it prints nicely

**EXAMPLES:**
Combine Values into a Vector or List

install_packages(package_name)

Install an R package into Sage’s R installation.

EXAMPLES:

```sage
sage: r.install_packages('aaMI')  # not tested # optional - rpy2
...
```

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
sage: r.library('grid')  # optional - rpy2
sage: 'grid' in r.eval('.packages()')  # optional - rpy2
True
sage: r.library('foobar')  # optional - rpy2
Traceback (most recent call last):
...
ImportError: ...
```

na()

Returns the NA in R.

OUTPUT: RElement – an element of R

EXAMPLES:

```sage
sage: r.na()  # optional - rpy2
[1] NA
```

plot(*args, **kwds)

The R plot function. Type r.help(‘plot’) for much more extensive documentation about this function. See also below for a brief introduction to more plotting with R.
If one simply wants to view an R graphic, using this function is sufficient (because it calls dev.off() to turn off the device).

However, if one wants to save the graphic to a specific file, it should be used as in the example below to write the output.

**EXAMPLES:**

This example saves a plot to the standard R output, usually a filename like Rplot001.png - from the command line, in the current directory, and in the cell directory in the notebook:

```sage
d=r.setwd('%S$%SAGE_TMP') # for doctesting only; ignore if you are trying this # optional - rpy2
sage: r.plot("1:10") # optional -- rgraphics # optional - rpy2
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
d=r.setwd('%S$%SAGE_TMP') # for doctesting only; ignore if you are trying this # optional - rpy2
sage: r.png(filename='%S$%SAGE_TMP')) # Note the double quotes in single quotes!
->; optional -- rgraphics # optional - rpy2
sage: filename = tmp_filename() + '.png' # optional - rpy2
sage: r.png(filename="%S"%filename) # filename not needed in notebook, used for doctesting; optional -- rgraphics # optional - rpy2
sage: x = r([1,2,3]) # optional - rpy2
sage: y = r([4,5,6]) # optional - rpy2
sage: r.plot(x,y) # optional -- rgraphics # optional - rpy2
null device 1
sage: import os; os.unlink(filename) # For doctesting, we remove the file; # optional -- rgraphics # optional - rpy2
```

Please note that for more extensive use of R’s plotting capabilities (such as the 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
d=r.setwd('%S$%SAGE_TMP') # for doctesting only; ignore if you are trying this # optional - rpy2
sage: r.X11() # not tested - opens interactive device on systems with X11 support # optional - rpy2
sage: r.quartz() # not tested - opens interactive device on OSX # optional - # rpy2
sage: r.hist("rnorm(100)") # not tested - makes a plot # optional - rpy2
sage: r.library("lattice") # not tested - loads R lattice plotting package # optional - rpy2
sage: r.histogram(x = "~ wt | cyl", data="mtcars") # not tested - makes a # lattice plot # optional - rpy2
sage: r.dev_off() # not tested, turns off the interactive viewer # optional - # rpy2
```

In the notebook, one can use `r.png()` to open the device, but would need to use the following since R lattice graphics do not automatically print away from the command line:

```sage
d=r.setwd('%S$%SAGE_TMP') # for doctesting only; ignore if you are trying this # optional - rpy2
sage: r.png(filename='%S$%SAGE_TMP')) # Note the double quotes in single quotes!
->; optional -- rgraphics # optional - rpy2
sage: filename = tmp_filename() + '.png' # Not needed in notebook, used for doctesting # optional - rpy2
sage: r.png(filename="%S"%filename) # filename not needed in notebook, used # for doctesting; optional -- rgraphics # optional - rpy2
null device 1
```

(continues on next page)
.. code:: sage

    sage: r.library("lattice")  # optional - rpy2
    sage: r("print(histogram(~wt | cyl, data=mtcars))") # plot should appear;
    "optional -- rgraphics # optional - rpy2"
    sage: import os; os.unlink(filename) # We remove the file for doctesting, not
    "needed in notebook; optional -- rgraphics # optional - rpy2"

.. code:: sage

    sage: r.png(filename="%s"%filename) # optional -- rgraphics #
    "optional - rpy2"
    NULL
    sage: x = r([1,2,3])  # optional - rpy2
    sage: y = r([4,5,6])  # optional - rpy2
    sage: r.plot(x,y)  # This saves to filename, but is not viewable from command
    "line; optional -- rgraphics # optional - rpy2"
    null device
    1
    sage: import os; os.unlink(filename) # We remove the file for doctesting;
    "optional -- rgraphics # optional - rpy2"

We want to make sure that we actually can view R graphics, which happens differently on different plat-
forms:

.. code:: sage

    sage: s = r.eval('capabilities("png")')  # Should be on Linux and Solaris #
    "optional - rpy2"
    sage: t = r.eval('capabilities("aqua")')  # Should be on all supported Mac
    "versions # optional - rpy2"
    sage: "TRUE" in s+t  # optional -- rgraphics # optional ~
    rpy2
    True

.. code:: sage

    sage: read(filename)
    Read filename into the R interpreter by calling R’s source function on a read-only file connection.

EXAMPLES:

.. code:: sage

    sage: filename = tmp_filename()  # optional - rpy2
    sage: f = open(filename, 'w')  # optional - rpy2
    sage: _ = f.write('a <- 2+2
    
    # optional -- rpy2
    sage: f.close()  # optional -- rpy2
    sage: r.read(filename)  # optional - rpy2
    sage: r.get('a')  # optional - rpy2
    '[1] 4'
**require**(*library_name*)

Load the library *library_name* into the R interpreter.

This function raises an ImportError if the given library is not known.

**INPUT:**

- *library_name* – string

**EXAMPLES:**

```python
sage: r.library('grid')  # optional - rpy2
sage: 'grid' in r.eval('(.packages())')  # optional - rpy2
True
sage: r.library('foobar')  # optional - rpy2
Traceback (most recent call last):
... ImportError: ...
```

**set**(*var*, *value*)

Set the variable *var* in R to what the string *value* evaluates to in R.

**INPUT:**

- *var* – a string
- *value* – a string

**EXAMPLES:**

```python
sage: r.set('a', '2 + 3')  # optional - rpy2
sage: r.get('a')  # optional - rpy2
'[1] 5'
```

**set_seed**(*seed=None*)

Set the seed for R interpreter.

The seed should be an integer.

**EXAMPLES:**

```python
sage: r = R()  # optional - rpy2
sage: r.set_seed(1)  # optional - rpy2
1
sage: r.sample('1:10', 5)  # random  # optional - rpy2
[1] 3 4 5 7 2
```

**source**(*s*)

Display the R source (if possible) about the function named *s*.

**INPUT:**

- *s* – a string representing the function whose source code you want to see

**OUTPUT:** string – source code

**EXAMPLES:**

```python
sage: print(r.source('c'))  # optional - rpy2
function (...) .Primitive('c')
```
version()

Return the version of R currently running.

OUTPUT: tuple of ints; string

EXAMPLES:

```python
sage: r.version() # not tested # optional - rpy2
((3, 0, 1), 'R version 3.0.1 (2013-05-16)')
sage: rint, rstr = r.version() # optional - rpy2
sage: rint[0] >= 3 # optional - rpy2
True
sage: rstr.startswith('R version') # optional - rpy2
True
```

class sage.interfaces.r.RElement(parent, value, is_name=False, name=None)

Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.interface.InterfaceElement

dot_product(other)

Implements the notation self . other.

INPUT:

• self, other – R elements

OUTPUT: R element

EXAMPLES:

```python
sage: c = r.c(1,2,3,4) # optional - rpy2
c.dot_product(c.t()) # optional - rpy2

[[1],
 [1, 1 2 3 4],
 [2, 2 4 6 8],
 [3, 3 6 9 12],
 [4, 4 8 12 16]]
sage: v = r([3,-1,8]) # optional - rpy2
sage: v.dot_product(v) # optional - rpy2

[[1],
 [74]]
```

is_string()

Tell whether this element is a string.

EXAMPLES:

```python
sage: r('"abc"').is_string() # optional - rpy2
True
sage: r([1,2,3]).is_string() # optional - rpy2
False
```

stat_model(x)

The tilde regression operator in R.

EXAMPLES:
\texttt{sage}: \texttt{x} = \texttt{r([1,2,3,4,5])} \quad \# \text{optional} - \texttt{rpy2} \\
\texttt{sage}: \texttt{y} = \texttt{r([3,5,7,9,11])} \quad \# \text{optional} - \texttt{rpy2} \\
\texttt{sage}: \texttt{a} = \texttt{r.lm( y.tilde(x) )} \quad \# \texttt{lm( y ~ x )} \quad \# \text{optional} - \texttt{rpy2} \\
\texttt{sage}: \texttt{d} = \texttt{a._sage_()} \quad \# \text{optional} - \texttt{rpy2} \\
\texttt{sage}: \texttt{d['DATA']['coefficients']['DATA'][1]} \quad \# \text{optional} - \texttt{rpy2} \\
2

\texttt{tilde(x)}

The tilde regression operator in \texttt{R}.

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage}: \texttt{x} = \texttt{r([1,2,3,4,5])} \quad \# \text{optional} - \texttt{rpy2} \\
\texttt{sage}: \texttt{y} = \texttt{r([3,5,7,9,11])} \quad \# \text{optional} - \texttt{rpy2} \\
\texttt{sage}: \texttt{a} = \texttt{r.lm( y.tilde(x) )} \quad \# \texttt{lm( y ~ x )} \quad \# \text{optional} - \texttt{rpy2} \\
\texttt{sage}: \texttt{d} = \texttt{a._sage_()} \quad \# \text{optional} - \texttt{rpy2} \\
\texttt{sage}: \texttt{d['DATA']['coefficients']['DATA'][1]} \quad \# \text{optional} - \texttt{rpy2} \\
\end{verbatim}

\textbf{class} \texttt{sage.interfaces.r.RFunction(parent, name, r_name=None)}

Bases: \texttt{sage.interfaces.interface.InterfaceFunction}

A Function in the \texttt{R} interface.

\textbf{INPUT:}

- parent – the \texttt{R} interface
- name – the name of the function for Python
- r_name – the name of the function in \texttt{R} itself (which can have dots in it)

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage}: \texttt{length = r.length} \quad \# \text{optional} - \texttt{rpy2} \\
\texttt{sage}: \texttt{type(length)} \quad \# \text{optional} - \texttt{rpy2} \\
<\texttt{class 'sage.interfaces.r.RFunction'>} \\
\texttt{sage}: \texttt{loads(dumps(length))} \quad \# \text{optional} - \texttt{rpy2} \\
\texttt{length}
\end{verbatim}

\textbf{class} \texttt{sage.interfaces.r.RFunctionElement(obj, name)}

Bases: \texttt{sage.interfaces.interface.InterfaceFunctionElement}

\texttt{sage.interfaces.r.is_RElement(x)}

Return True if \texttt{x} is an element in an \texttt{R} interface.

\textbf{INPUT:}

- \texttt{x} – object

\textbf{OUTPUT:} bool

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage}: \texttt{from sage.interfaces.r import is_RElement} \quad \# \text{optional} - \texttt{rpy2} \\
\texttt{sage}: \texttt{is_RElement(2)} \quad \# \text{optional} - \texttt{rpy2} \\
\texttt{False} \\
\texttt{sage}: \texttt{is_RElement(r(2))} \quad \# \text{optional} - \texttt{rpy2} \\
\texttt{True}
\end{verbatim}
sage.interfaces.r.r_console()
Spawn a new R command-line session.

EXAMPLES:

```python
sage: r.console()  # not tested # optional - rpy2
R version 2.6.1 (2007-11-26)
Copyright (C) 2007 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
...
```

sage.interfaces.r.r_version()
Return the R version.

EXAMPLES:

```python
sage: r_version()  # not tested # optional - rpy2
((3, 0, 1), 'R version 3.0.1 (2013-05-16)')
sage: rint, rstr = r_version()  # optional - rpy2
sage: rint[0] >= 3  # optional - rpy2
True
sage: rstr.startswith('R version')  # optional - rpy2
True
```

sage.interfaces.r.reduce_load_R()
Used for reconstructing a copy of the R interpreter from a pickle.

EXAMPLES:

```python
sage: from sage.interfaces.r import reduce_load_R  # optional - rpy2
sage: reduce_load_R()  # optional - rpy2
R Interpreter
```
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)

```python
class sage.interfaces.rubik.CubexSolver
    Bases: object

    format_cube(facets)
    solve(facets)

EXAMPLES:

sage: from sage.interfaces.rubik import *
      # optional - rubiks
sage: C = RubiksCube("R U")
      # optional - rubiks
sage: CubexSolver().solve(C.facets())
      # optional - rubiks
'R U'
sage: C = RubiksCube("R U F L B D")
      # optional - rubiks
sage: sol = CubexSolver().solve(C.facets()); sol
      # optional - rubiks
sage: RubiksCube(sol) == C
      # optional - rubiks
True
sage: C = RubiksCube("R2 F'")
      # optional - rubiks
sage: CubexSolver().solve(C.facets())
      # optional - rubiks
"R' R' F'"
sage: C = RubiksCube().scramble()
      # optional - rubiks
sage: sol = CubexSolver().solve(C.facets())
      # optional - rubiks
sage: C == RubiksCube(sol)
      # optional - rubiks
True
"R fun"
class sage.interfaces.rubik.DikSolver
    Bases: object

    format_cube(facets)

    solve(facets, timeout=10, extra_time=2)

    EXAMPLES:
    sage: from sage.interfaces.rubik import *  # optional - rubiks
    sage: C = RubiksCube().move("R U")        # optional - rubiks
    sage: DikSolver().solve(C.facets())       # optional - rubiks
       'R U'
    sage: C = RubiksCube().move("R U F L B D") # optional - rubiks
    sage: DikSolver().solve(C.facets())       # optional - rubiks
       'R U F L B D'
    sage: C = RubiksCube().move("R2 F")      # optional - rubiks
    sage: DikSolver().solve(C.facets())       # optional - rubiks
       "R2 F"

class sage.interfaces.rubik.OptimalSolver(verbos=False, wait=True)
    Bases: object

    Interface to Michael Reid’s optimal Rubik’s Cube solver.

    format_cube(facets)

    ready()

    solve(facets)

    The initial startup and precomputation are substantial...

    Todo: Let it keep searching once it found a solution?

    EXAMPLES:
    sage: from sage.interfaces.rubik import *     # optional - rubiks
    sage: solver = DikSolver()                    # optional - rubiks
    sage: solver = OptimalSolver()                # optional - rubiks # long time (28s on sage.˓
    →math, 2012)
       Initializing tables...
       Done.
    sage: C = RubiksCube("R U")                 # optional - rubiks
    sage: solver.solve(C.facets())               # optional - rubiks
       'R U'
    sage: C = RubiksCube("R U F L B D")         # optional - rubiks
    sage: solver.solve(C.facets())               # optional - rubiks
       'R U F L B D'
    sage: C = RubiksCube("R2 D2")               # optional - rubiks
    sage: solver.solve(C.facets())               # optional - rubiks
       'R2 D2'

    start()

    stop()

class sage.interfaces.rubik.SingNot(s)
    Bases: object
This class is to resolve difference between various Singmaster notation.
Case is ignored, and the second and third letters may be swapped.

EXAMPLES:

```python
sage: from sage.interfaces.rubik import SingNot
sage: SingNot("acb") == SingNot("ACB")
True
sage: SingNot("acb") == SingNot("bca")
False
```
This is an expect interface to another copy of the Sage interpreter.

```python
class sage.interfaces.sage0.Sage(logfile=None, preparse=True, python=False, init_code=None,
                                 server=None, server_tmpdir=None, remote_cleaner=True, **kwds):
    Bases: sage.interfaces.tab_completion.ExtraTabCompletion, 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.

```python
sage: s = Sage()
```

Create the element 2 in our new copy of Sage, and cube it.

```python
sage: a = s(2)
sage: a^3
8
```

Create a vector space of dimension 4, and compute its generators:

```python
sage: V = s('QQ^4')
sage: V.gens()
((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))
```

Note that V is not a vector space, it’s a wrapper around an object (which happens to be a vector space), in another running instance of Sage.

```python
sage: type(V)
<class 'sage.interfaces.sage0.SageElement'>
sage: V.parent()
Sage
sage: g = V.0; g
(1, 0, 0, 0)
```

(continues on next page)
sage: g.parent()
Sage

We can still get the actual parent by using the name attribute of g, which is the variable name of the object in the child process.

sage: s('%s.parent() % g.name()')
Vector space of dimension 4 over Rational Field

Note that the memory space is completely different.

sage: x = 10
sage: s('x = 5')
5
sage: x
10
sage: s('x')
5

We can have the child interpreter itself make another child Sage process, so now three copies of Sage are running:

sage: s3 = s('Sage()')
sage: a = s3(10)
sage: a
10

This $a = 10$ is in a subprocess of a subprocess of your original Sage.

sage: _ = s.eval('%s.eval("x=8") % s3.name()')
sage: s3('"x"')
8
sage: s('x')
5
sage: x
10

The double quotes are needed because the call to s3 first evaluates its arguments using the s interpreter, so the call to s3 is passed s('"x"'), which is the string "x" in the s interpreter.

clear(var)
Clear the variable named var.

Note that the exact format of the NameError for a cleared variable is slightly platform dependent, see trac ticket #10539.

EXAMPLES:

sage: sage0.set('x', '2')
sage: sage0.get('x')
'2'
sage: sage0.clear('x')
sage: 'NameError' in sage0.get('x')
True
console()
Spawn a new Sage command-line session.

EXAMPLES:

```
sage: sage0.console() #not tested
| SageMath version ..., Release Date: ...
| Using Python .... Type "help()" for help.
```

... 

cputime(t=None)
Return cputime since this Sage subprocess was started.

EXAMPLES:

```
sage: sage0.cputime() # random output
1.3530439999999999
sage: sage0('factor(2^157-1)')
852133201 * 60726444167 * 1654058017289 * 2134387368610417
sage: sage0.cputime() # random output
1.6462939999999999
```

eval(line, strip=True, **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'
```

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

EXAMPLES:

```
sage: sage0.set('x', '2')
sage: sage0.get('x')
'2'
```

new(x)
EXAMPLES:

```
sage: sage0.new(2)
2
sage: _.parent()
Sage
```

preparse(x)
Returns the preparsed version of the string s.
EXAMPLES:

```python
sage: sage0.preparse('2+2')
'Integer(2)+Integer(2)'
```

**set(var, value)**

Set the variable `var` to the given value.

EXAMPLES:

```python
sage: sage0.set('x', '2')
sage: sage0.get('x')
'2'
```

**version()**

EXAMPLES:

```python
sage: sage0.version()
'SageMath version ..., Release Date: ...'
sage: sage0.version() == version()
True
```

```python
sage.interfaces.sage0.SageElement(parent, value, is_name=False, name=None)
Bases: sage.interfaces.expect.ExpectElement
```

```python
sage.interfaces.sage0.SageFunction(obj, name)
Bases: sage.interfaces.expect.FunctionElement
```

```python
sage.interfaces.sage0.reduce_load_Sage()
EXAMPLES:

```python
sage: from sage.interfaces.sage0 import reduce_load_Sage
sage: reduce_load_Sage()
Sage
```

```python
sage.interfaces.sage0.reduce_load_element(s)
EXAMPLES:

```python
sage: from sage.interfaces.sage0 import reduce_load_element
sage: s = dumps(1/2)
sage: half = reduce_load_element(s); half
1/2
sage: half.parent()
Sage
```

```python
sage.interfaces.sage0.sage0_console()
Spawn a new Sage command-line session.

EXAMPLES:

```python
sage: sage0_console() #not tested
```

sage.interfaces.sage0.sage0_version()
EXAMPLES:

```python
sage: from sage.interfaces.sage0 import sage0_version
sage: sage0_version() == version()
True
```
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:**

```plaintext
sage: scilab.eval('2+2')  
'ans =
\n 4.'

sage: scilab('2+2')  
4.

sage: a = scilab(10)  
sage: a**10  
1.000D+10
```

Tutorial based on the MATLAB interface tutorial:

**EXAMPLES:**

```plaintext
sage: scilab('4+10')  
14.

sage: scilab('date')  
15-Feb-2010

sage: scilab('5*10 + 6')  
56.

sage: scilab('(6+6)/3')  
4.

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: my_scalar = scilab('3.1415')  
sage: my_scalar  
3.1415
```

(continues on next page)
sage: my_vector1 = scilab('[1,5,7]')  # optional - scilab
1.  5.  7.
sage: my_vector2 = scilab('[1;5;7]')  # optional - scilab
1.  5.  7.
sage: my_vector1 * my_vector2  # optional - scilab
75.

sage: row_vector1 = scilab('[1 2 3]')  # optional - scilab
1.  2.  3.
sage: row_vector2 = scilab('[3 2 1]')  # optional - scilab
3.  2.  1.
sage: matrix_from_row_vec = scilab('[%s; %s]'%(row_vector1.name(), row_vector2.name()))  # optional - scilab
sage: matrix_from_row_vec  # optional - scilab
1.  2.  3.  3.  2.  1.

sage: column_vector1 = scilab('[1;3]')  # optional - scilab
1.  3.
sage: column_vector2 = scilab('[2;8]')  # optional - scilab
2.  8.
sage: matrix_from_col_vec = scilab('[%s %s]'%(column_vector1.name(), column_vector2.name()))  # optional - scilab
sage: matrix_from_col_vec  # optional - scilab
1.  2.  3.  8.  1.  1.

sage: my_matrix = scilab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]')  # optional - scilab
sage: my_matrix  # optional - scilab
8.  12.  19.
7.  3.  2.
12.  4.  23.
8.  1.  1.

sage: combined_matrix = scilab('[%s, %s]'%(my_matrix.name(), my_matrix.name()))  # optional - scilab
sage: combined_matrix  # optional - scilab
7.  3.  2.  7.  3.  2.
12.  4.  23.  12.  4.  23.
8.  1.  1.  8.  1.  1.

sage: tm = scilab('0.5:2:10')  # optional - scilab
sage: tm  # optional - scilab
0.5  2.5  4.5  6.5  8.5

sage: my_vector1 = scilab('[1,5,7]')  # optional - scilab
1.
sage: my_vector1[1]  # optional - scilab
1.
Matrix indexing works as follows:

```python
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
4.
```

One can also use square brackets:

```python
sage: my_matrix[3,2]  # optional - scilab
4.
```

Setting using parenthesis cannot work (because of how the Python language works). Use square brackets or the set function:

```python
sage: my_matrix = scilab(['[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]'])  # optional - scilab
sage: my_matrix.set(2,3, 1999)  # optional - scilab
sage: my_matrix  # optional - scilab
8. 12. 19.
7. 3. 1999.
12. 4. 23.
8. 1. 1.
```

```python
sage: my_matrix[2,3] = -126  # optional - scilab
sage: my_matrix  # optional - scilab
8. 12. 19.
7. 3. -126.
12. 4. 23.
8. 1. 1.
```

AUTHORS:


**class** `sage.interfaces.scilab.Scilab(maxread=None, script_subdirectory=None, logfile=None, server=None, server_tmpdir=None, seed=None)`

Bases: `sage.interfaces.expect.Expect`

Interface to the Scilab interpreter.

**EXAMPLES:**

```python
sage: a = scilab('[ 1, 1, 2; 3, 5, 8; 13, 21, 33 ]')  # optional - scilab
sage: b = scilab('[ 1; 3; 13]')  # optional - scilab
sage: c = a * b  # optional - scilab
sage: print(c)  # optional - scilab
30.
122.
505.
```

```
console()  
Starts Scilab console.
```
EXAMPLES:

```python
sage: scilab.console()  # optional - scilab; not tested
```

`eval(command, *args, **kwds)`

Evaluates commands.

EXAMPLES:

```python
sage: scilab.eval("5")  # optional - scilab
'ans = 5.'
sage: scilab.eval("d=44")  # optional - scilab
'd = 44.'
```

`get(var)`

Get the value of the variable var.

EXAMPLES:

```python
sage: scilab.eval("b=124;")  # optional - scilab
'sage: scilab.get("b")  # optional - scilab
'124.'
```

`sage2scilab_matrix_string(A)`

Return a Scilab matrix from a Sage matrix.

**INPUT:** A Sage matrix with entries in the rationals or reals.

**OUTPUT:** A string that evaluates to an Scilab matrix.

EXAMPLES:

```python
sage: M33 = MatrixSpace(QQ,3,3)  # optional - scilab
sage: A = M33([1,2,3,4,5,6,7,8,0])  # optional - scilab
sage: scilab.sage2scilab_matrix_string(A)  # optional - scilab
'[[1, 2, 3; 4, 5, 6; 7, 8, 0]'
```

`set(var, value)`

Set the variable var to the given value.

EXAMPLES:

```python
sage: scilab.set("a", 123)  # optional - scilab
sage: scilab.get("a")  # optional - scilab
'123.'
```
**set_seed** *(seed=None)*
Set the seed for gp interpreter.

The seed should be an integer.

**EXAMPLES:**

```
sage: from sage.interfaces.scilab import Scilab  # optional - scilab
sage: s = Scilab()  # optional - scilab
sage: s.set_seed(1)  # optional - scilab
1
sage: [s.rand() for i in range(5)]  # optional - scilab
[0.6040239, 0.0079647, 0.6643966, 0.9832111, 0.5321420]
```

**version()**
Returns the version of the Scilab software used.

**EXAMPLES:**

```
sage: scilab.version()  # optional - scilab
'scilab-...'
```

**whos** *(name=None, typ=None)*
Returns information about current objects. Arguments: nam: first characters of selected names typ: name of selected Scilab variable type

**EXAMPLES:**

```
sage: scilab.whos("core")  # optional - scilab
'Name    Type    Size    Bytes...'  
Name    Type    Size    Bytes...'  
'sage: scilab.whos(typ='function')  # optional - scilab
'Name    Type    Size    Bytes...'  
'type    function    Size    Bytes...'
```

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

**set** *(i, j, x)*
Set the variable var to the given value.

**EXAMPLES:**

```
sage: scilab.set('c', 125)  # optional - scilab
'sage: scilab.get('c')  # optional - scilab
125.'
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:

```python
sage: from sage.interfaces.scilab import scilab_console # optional - scilab
sage: scilab_console()                                   # optional - scilab; not tested
```

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:

```python
sage: from sage.interfaces.scilab import scilab_version # optional - scilab
sage: scilab_version()                                  # optional - scilab
'scilab-...'"
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
sage: R1 = singular.ring(0, '(x,y)', 'dp')
sage: R1
polynomial ring, over a field, global ordering
// coefficients: QQ
// number of vars : 2
// block 1 : ordering dp
// : names x y
// block 2 : ordering C
sage: f = singular('9x16 - 18x13y2 - 9x12y3 + 9x10y4 - 18x11y2 + 36x8y4 + 18x7y5 -
˓→18x5y6 + 9x6y4 - 18x3y6 - 9x2y7 + 9y8')
sage: f
9*x^16-18*x^13*y^2-9*x^12*y^3+9*x^10*y^4-18*x^11*y^2+36*x^8*y^4+18*x^7*y^5-18*x^5*y^6+9*x^6*y^4-18*x^3*y^6-9*x^2*y^7+9*y^8

sage: f.parent()
Singular

sage: F = f.factorize(); F
[[1]:
  [1]=9
  [2]=x^6-2*x^3*y^2-x^2*y^3+y^4
  [3]=-x^5+y^2
[2]:
  1,1,2]

sage: F[1][2].sage()
x^6 - 2*x^3*y^2 - x^2*y^3 + y^4

sage: g = F[1][2].sage()
```

We can convert $f$ and each exponent back to Sage objects as well.

```sage
sage: g
x^6 - 2*x^3*y^2 - x^2*y^3 + y^4

sage: g.parent()
Multivariate Polynomial Ring in x, y over Rational Field
```

This example illustrates polynomial GCD’s:

```sage
sage: R2 = singular.ring(0, '(x,y,z)', 'lp')
sage: a = singular.new('3x^2*(x+y)')
sage: b = singular.new('9x*(y^2-x^2)')
sage: g = a.gcd(b)
sage: g
x^2+x*y
```
This example illustrates computation of a Groebner basis:

```
sage: R3 = singular.ring(0, '(a,b,c,d)', 'lp')
sage: I = singular.ideal(['a + b + c + d', 'a*b + a*d + b*c + c*d', 'a*b*c + a*b*d + a*c*d + b*c*d', 'a*b*c*d - 1'])
sage: I2 = I.groebner()
sage: I2

\begin{align*}
&c^2d^6-c^2d^2-d^4+1, \\
c^3d^2+c^2d^3-c-d \\
b*d^4-b+d^5-d, \\
b+c-b*d^5+c^2d^4+c*d-d^6-d^2, \\
b^2+2b^2d+d^2, \\
a+b+c+d
\end{align*}
```

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

\begin{align*}
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
\end{align*}
```

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('x^3+y^3+(x-y)*x^2*y^2+z^2')
sage: f

\begin{align*}
x^3*y^2-x^2*y^3+x^3+y^3+z^2
\end{align*}
```

```
sage: R1 = singular.ring(0, '(x,y,z)', 'ds')
sage: f = R.fetch(f)
sage: f

\begin{align*}
z^2+x^3+y^3+x^3*y^2-x^2*y^3
\end{align*}
```

We can calculate the Milnor number of $f$:

```
sage: _ = singular.LIB('sing.lib')  # assign to _ to suppress printing
sage: f.milnor()

4
```

The Jacobian applied twice yields the Hessian matrix of $f$, with which we can compute.

```
sage: H = f.jacob().jacob()
sage: H

\begin{align*}
6*x+6*x^2*y^2-2*y^3, 6*x^2*y-6*x^2*y^2, 0, \\
6*x^2*y-6*x^2*y^2, 6*y+2*x^3-6*x^2*y, 0, \\
0, 0, 2
\end{align*}
```
The 1x1 and 2x2 minors:

\begin{verbatim}
sage: H.minor(1)
2,
6*y+2*x^3-6*x^2*y,
6*x^2*y-6*x*y^2,
6*x^2*y-6*x*y^2,
6*x+6*x*y^2-2*y^3,
0,
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
\end{verbatim}

\begin{verbatim}
sage: _=singular.eval('option(redSB)')
sage: H.minor(1).groebner()
1
\end{verbatim}

## 44.3 Computing the Genus

We compute the projective genus of ideals that define curves over \( \mathbb{Q} \). It is very important to load the `normal.lib` library before calling the `genus` command, or you'll get an error message.

**EXAMPLES:**

\begin{verbatim}
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
\end{verbatim}

Note that the genus can be much smaller than the degree:
44.4 An Important Concept

AUTHORS:

- Neal Harris

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.

```python
sage: singular.lib('polylib.lib')
sage: singular.lib('polylib.lib'); R = singular.lib('polylib.lib'); R

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:
```

We restart everything and try again, but correctly.

```python
sage: singular.quit()
sage: singular.lib('polylib.lib'); R = singular.lib('polylib.lib'); R
sage: I = singular.ideal('cyclic(6)')
sage: I.groebner()
... f^48-2554*f^42-15674*f^36+12326*f^30-12326*f^18+15674*f^12+2554*f^6-1,
```

It's important to understand why the first attempt at computing a basis failed. The line where we gave Singular the input ‘groebner(I)’ was useless because Singular has no idea what ‘I’ is! Although ‘I’ is an object that we computed with calls to Singular functions, it actually lives in Sage. As a consequence, the name ‘I’ means nothing to Singular. When we called I.groebner(), Sage was able to call the groebner function on ‘I’ in Singular, since ‘I’ actually means something to Sage.
44.5 Long Input

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

```sage
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)
```

**class** `sage.interfaces.singular.Singular`

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

Interface to the Singular interpreter.

EXAMPLES: A Groebner basis example.
```

```sage
sage: R = singular.ring(0, '(x0,x1,x2)', 'lp')
sage: I = singular.ideal([ 'x0*x1*x2 -x0^2*x2', 'x0^2*x1*x2-x0*x1^2*x2-x0*x1*x2^2', 'x0*x1-x0*x2-x1*x2'])
sage: I.groebner()
x1^2*x2^2, x0*x2^3-x1^2*x2^2+x1*x2^3, x0*x1-x0*x2-x1*x2, x0^2*x2-x0*x2^2-x1*x2^2
```

AUTHORS:
- David Joyner and William Stein

**LIB**(lib, reload=False)

Load the Singular library named lib.

Note that if the library was already loaded during this session it is not reloaded unless the optional reload argument is True (the default is False).

**EXAMPLES:**

```sage
sage: singular.lib('sing.lib')
sage: singular.lib('sing.lib', reload=True)
```

**clear**(var)

Clear the variable named var.

**EXAMPLES:**

```sage
sage: singular.set('int', 'x', '2')
sage: singular.get('x')
'2'
sage: singular.clear('x')
```

“Clearing the variable” means to allow to free the memory that it uses in the Singular sub-process. However, the actual deletion of the variable is only committed when the next element in the Singular interface is created:
sage: singular.get('x')
2
sage: a = singular(3)
sage: singular.get('x')
'x'

console()

EXAMPLES:

sage: singular_console() #not tested
SINGULAR / Development
A Computer Algebra System for Polynomial Computations / version 3-0-4

by: G.-M. Greuel, G. Pfister, H. Schoenemann  
FB Mathematik der Universitaet, D-67653 Kaiserslautern   

sage: t = singular.cputime()
sage: R = singular.ring(0, '(x0,x1,x2)', 'lp')
sage: I = singular.ideal(['x0*x1*x2-x0^2*x2', 'x0^2*x1*x2-x0*x1^2*x2-x0*x1*x2^2', 'x0*x1-x0*x2-x1*x2'])
sage: gb = I.groebner()
sage: singular.cputime(t) #random
0.02

current_ring()

Returns the current ring of the running Singular session.

EXAMPLES:

sage: r = PolynomialRing(GF(127),3,'xyz', order='invlex')
sage: r._singular_()
polynomial ring, over a field, global ordering
// coefficients: ZZ/127
// number of vars : 3
//   block 1 : ordering rp
//       : names  x y z
//   block 2 : ordering C

sage: singular.current_ring()
polynomial ring, over a field, global ordering
// coefficients: ZZ/127
// number of vars : 3
//   block 1 : ordering rp
//       : names  x y z
//   block 2 : ordering C

current_ring_name()

Returns the Singular name of the currently active ring in Singular.

OUTPUT: currently active ring’s name
EXAMPLES:

```python
sage: r = PolynomialRing(GF(127),3,'xyz')
sage: 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:**

```python
sage: singular.eval('2 > 1')
'1'
sage: singular.eval('2 + 2')
'4'
```

if the verbosity level is > 1 comments are also printed and not only returned.

```python
sage: r = singular.ring(0,'(x,y,z)','dp')
sage: i = singular.ideal(['x^2','y^2','z^2'])
sage: s = i.std()
sage: singular.eval('hilb(%s)'%(s.name()))
"// 1 t^0\n// -3 t^2\n// 3 t^4\n// -1 t^6\n// 1 t^8\n// 3 t^10\n// 3 t^12\n// 1 t^14\n// dimension (affine) = 0\n// degree (affine) = 8"
```

```python
sage: from sage.misc.verbose import set_verbose
sage: set_verbose(1)
sage: o = singular.eval('hilb(%s)'%(s.name()))
"// 1 t^0
// -3 t^2
// 3 t^4
// -1 t^6
// 1 t^8
// 3 t^10
// 3 t^12
// 1 t^14
// 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.

```python
sage: o = s.hilb()
"// 1 t^0
// -3 t^2
// 3 t^4"
```

(continues on next page)
Sage 9.5 Reference Manual: Interpreter Interfaces, Release 9.5

---

(continued from previous page)

```plaintext
// -1 t^6
// 1 t^0
// 3 t^1
// 3 t^2
// 1 t^3
// dimension (affine) = 0
// degree (affine) = 8
// ** right side is not a datum, assignment ignored
...  
```

rather than ignored

```plaintext
sage: set_verbose(0)
sage: o = s.hilb()
```

### `get(var)`

Get string representation of variable named var.

**EXAMPLES:**

```plaintext
sage: singular.set('int', 'x', '2')
sage: singular.get('x')
'2'
```

### `ideal(*gens)`

Return the ideal generated by gens.

**INPUT:**

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

**OUTPUT:** the Singular ideal generated by the given list of gens

**EXAMPLES:** A Groebner basis example done in a different way.

```plaintext
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
```

```plaintext
sage: i2 = singular.ideal('groebner(%s);%i1.name())
sage: i2
x1^2*x2^2, x0^2*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:**

---

44.5. Long Input 367
list(x)

Creates 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]:
  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"])  # Escape the string
[1]:
  1
[2]:
  2
[3]:
  hi
```

load(lib, reload=False)

Load the Singular library named lib.

Note that if the library was already loaded during this session it is not reloaded unless the optional reload argument is True (the default is False).

EXAMPLES:
matrix(nrows, ncols, entries=None)

EXAMPLES:

```python
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
```

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:

```python
sage: singular.option()
//options: redefine loadLib usage prompt
sage: singular.option('get')
0, 10321
sage: old_options = _
sage: singular.option('noredefine')
sage: singular.option()
//options: loadLib usage prompt
sage: singular.option('set', old_options)
sage: singular.option('get')
0, 10321
```

ring(char=0, vars='(x)', order='lp', check=True)

Create a Singular ring and makes it the current ring.

INPUT:

- char - characteristic of the base ring (see examples below), which must be either 0, prime (!), or one of several special codes (see examples below).
- vars - a tuple or string that defines the variable names
• **order** - string - the monomial order (default: ‘lp’)
• **check** - if True, check primality of the characteristic if it is an integer.

**OUTPUT:** a Singular ring

**Note:** This function is not identical to calling the Singular ring function. In particular, it also attempts to "kill" the variable names, so they can actually be used without getting errors, and it sets printing of elements for this range to short (i.e., with *'s and carets).

**EXAMPLES:** We first declare \( \mathbb{Q}[x,y,z] \) with degree reverse lexicographic ordering.

```
sage: R = singular.ring(0, '(x,y,z)', 'dp')
sage: R
polynomial ring, over a field, global ordering
// coefficients: QQ
// number of vars : 3
// block 1 : ordering dp
// : names x y z
// block 2 : ordering C
```

```
sage: R1 = singular.ring(32003, '(x,y,z)', 'dp')
sage: R2 = singular.ring(32003, '(a,b,c,d)', 'lp')
```

This is a ring in variables named x(1) through x(10) over the finite field of order 7:

```
sage: R3 = singular.ring(7, '(x(1..10))', 'ds')
```

This is a polynomial ring over the transcendental extension \( \mathbb{Q}(a) \) of \( \mathbb{Q} \):

```
sage: R4 = singular.ring('(/0,a)', '(mu,nu)', 'lp')
```

This is a ring over the field of single-precision floats:

```
sage: R5 = singular.ring('real', '(a,b)', 'lp')
```

This is over 50-digit floats:

```
sage: R6 = singular.ring('(/real,50)', '(a,b)', 'lp')
sage: R7 = singular.ring('(/complex,50,i)', '(a,b)', 'lp')
```

To use a ring that you’ve defined, use the set_ring() method on the ring. This sets the ring to be the “current ring”. For example,

```
sage: R = singular.ring(7, '(a,b)', 'ds')
sage: S = singular.ring('real', '(a,b)', 'lp')
sage: singular.new('10*a')
(1.000e+01)*a
sage: R.set_ring()
sage: singular.new('10*a')
3*a
```

**set**(type, name, value)
Set the variable with given name to the given value.
REMARK:

If a variable in the Singular interface was previously marked for deletion, the actual deletion is done here, before the new variable is created in Singular.

EXAMPLES:

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

We test that an unused variable is only actually deleted if this method is called:

```
sage: a = singular(3)
sage: n = a.name()
sage: del a
sage: singular.eval(n)
'3'
sage: singular.set('int', 'y', '5')
sage: singular.eval('defined(%s)'%n)
'0'
```

**set_ring(R)**

Sets the current Singular ring to R.

EXAMPLES:

```
sage: R = singular.ring(7, '(a,b)', 'ds')
sage: S = singular.ring('real', '(a,b)', 'lp')
sage: singular.current_ring()
polynomial ring, over a field, global ordering
  //  coefficients: Float()
  //  number of vars : 2
  //     block 1 : ordering lp
  //       : names   a b
  //     block 2 : ordering C
sage: singular.set_ring(R)
sage: singular.current_ring()
polynomial ring, over a field, local ordering
  //  coefficients: ZZ/7
  //  number of vars : 2
  //     block 1 : ordering ds
  //       : names   a b
  //     block 2 : ordering C
```

**set_seed**(seed=None)

Set the seed for singular interpreter.

The seed should be an integer at least 1 and not more than 30 bits. See http://www.singular.uni-kl.de/Manual/html/sing_19.htm#SEC26 and http://www.singular.uni-kl.de/Manual/html/sing_283.htm#SEC323

EXAMPLES:

```
sage: s = Singular()
sage: s.set_seed(1)
```

(continues on next page)
1
sage: [s.random(1,10) for i in range(5)]
[8, 10, 4, 9, 1]

**setring**\( (R) \)
Sets the current Singular ring to \( R \).

**EXAMPLES:**

```python
sage: R = singular.ring(7, '(a,b)', 'ds')
sage: S = singular.ring('real', '(a,b)', 'lp')
sage: singular.current_ring()
polynomial ring, over a field, global ordering
// coefficients: Float()
// number of vars : 2
//   block 1: ordering lp
//       : names a b
//   block 2: ordering C
sage: singular.set_ring(R)
sage: singular.current_ring()
polynomial ring, over a field, local ordering
// coefficients: ZZ/7
// number of vars : 2
//   block 1: ordering ds
//       : names a b
//   block 2: ordering C
```

**string**\( (x) \)
Creates a Singular string from a Sage string. Note that the Sage string has to be “double-quoted”.

**EXAMPLES:**

```python
sage: singular.string("Sage")
Sage
```

**version**\()\)
Return the version of Singular being used.

**EXAMPLES:**

```python
sage: singular.version()
"Singular ... version 4...
```

**class**\( \text{sage.interfaces.singular.SingularElement} \)(parent, type, value, is_name=False)
Bases: \text{sage.interfaces.tab_completion.ExtraTabCompletion}, \text{sage.interfaces.expect.ExpectElement}

**EXAMPLES:**

```python
sage: a = singular(2)
sage: loads(dumps(a))
2
```

**attrib**\( (name, value=None) \)
Get and set attributes for self.
INPUT:

- name - string to choose the attribute
- value - boolean value or None for reading, (default: None)

VALUES:
- isSB - the standard basis property is set by all commands computing a standard basis like groebner, std, stdhilb etc.; used by lift, dim, degree, mult, hilb, vdim, kbase
- isHomog - the weight vector for homogeneous or quasihomogeneous ideals/modules
- isCI - complete intersection property
- isCM - Cohen-Macaulay property
- rank - set the rank of a module (see nrows) with SB - value of type ideal, resp. module, is std with Hilb - value of type intvec is hilb(_,1) (see hilb) with Res - value of type list is a free resolution with Dim - value of type int is the dimension (see dim) with Mult - value of type int is the multiplicity (see mult)

EXAMPLES:

```python
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(%)' % Ibar.name()) # sage7 name is random
// ** sage7 is no standard basis
4
sage: Ibar.attrib('isSB',1)
sage: singular.eval('vdim(%)' % Ibar.name())
'4'
```

`is_string()`

Tell whether this element is a string.

EXAMPLES:

```python
sage: singular('"abc"').is_string()
True
sage: singular('1').is_string()
False
```

`sage_flattened_str_list()`

EXAMPLES:

```python
sage: R=singular.ring(0,'(x,y)','dp')
sage: RL = R.ringlist()
sage: RL.sage_flattened_str_list()
['0', 'x', 'y', 'dp', '1,1', 'C', '0', '_[1]=0']
```

`sage_global_ring()`

Return the current basering in Singular as a polynomial ring or quotient ring.

EXAMPLES:

```python
sage: singular.eval('ring rl = (9,x),(a,b,c,d,e,f),(M((1,2,3,0)),wp(2,3),lp)')
''
sage: R = singular('rl').sage_global_ring()
sage: R
Multivariate Polynomial Ring in a, b, c, d, e, f over Finite Field in x of size 3^2
```

(continues on next page)
sage: R.term_order()
Block term order with blocks:
(Matrix term order with matrix
[1 2]
[3 0],
Weighted degree reverse lexicographic term order with weights (2, 3),
Lexicographic term order of length 2)

sage: singular.eval('ring r2 = (0,x),(a,b,c),dp')
''
sage: singular('r2').sage_global_ring()
Multivariate Polynomial Ring in a, b, c over Fraction Field of Univariate
→Polynomial Ring in x over Rational Field

sage: singular.eval('ring r3 = (3,z),(a,b,c),dp')
''
sage: singular.eval('minpoly = 1+z+z2+z3+z4')
''
sage: singular('r3').sage_global_ring()
Multivariate Polynomial Ring in a, b, c over Finite Field in z of size 3^4

Real and complex fields in both Singular and Sage are defined with a precision. The precision in Singular is
given in terms of digits, but in Sage it is given in terms of bits. So, the digit precision is internally converted
to a reasonable bit precision:

sage: singular.eval('ring r4 = (real,20),(a,b,c),dp')
''
sage: singular('r4').sage_global_ring()
Multivariate Polynomial Ring in a, b, c over Real Field with 70 bits of
→precision

The case of complex coefficients is not fully supported, yet, since the generator of a complex field in Sage
is always called “I”:

sage: singular.eval('ring r5 = (complex,15,j),(a,b,c),dp')
''
sage: R = singular('r5').sage_global_ring(); R
Multivariate Polynomial Ring in a, b, c over Complex Field with 54 bits of
→precision
sage: R.base_ring()('k')
Traceback (most recent call last):
...
ValueError: given string 'k' is not a complex number
sage: R.base_ring()('I')
1.00000000000000*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')
''

(continues on next page)
In our last example, the base ring is a quotient ring:

```
sage: Q = singular(std(ideal(x^2,x+y^2+z^3)), type='qring')
sage: Q.sage_global_ring()
Quotient of Multivariate Polynomial Ring in x, y, z over Finite Field in a of size 3^2 by the ideal (y^4 - y^2*z^3 + z^6, x + y^2 + z^3)
```

AUTHOR:
• Simon King (2011-06-06)

`sage_matrix(R, sparse=True)`
Returns Sage matrix for self

INPUT:
• R - (default: None); an optional ring, over which the resulting matrix is going to be defined. By default, the output of `sage_global_ring()` is used.
• sparse - (default: True); determines whether the resulting matrix is sparse or not.

EXAMPLES:

```
sage: R = singular.ring(0, '(x,y,z)', 'dp')
sage: A = singular.matrix(2,2)
sage: A.sage_matrix(ZZ)
[0 0]
[0 0]
sage: A.sage_matrix(RDF)
[0.0 0.0]
[0.0 0.0]
```

`sage_poly(R=None, kcache=None)`
Returns a Sage polynomial in the ring r matching the provided poly which is a singular polynomial.

INPUT:
• R - (default: None); an optional polynomial ring. If it is provided, then you have to make sure that it matches the current singular ring as, e.g., returned by `singular.current_ring()`. By default, the output of `sage_global_ring()` is used.
• kcache - (default: None); an optional dictionary for faster finite field lookups, this is mainly useful for finite extension fields

OUTPUT: MPolynomial

EXAMPLES:

```
sage: R = PolynomialRing(GF(2^8, 'a'), 'x,y')
sage: f = R('a^20*x^2*y+a^10+x')
sage: f._singular_.sage_poly(R) == f
True
```
sage: R = PolynomialRing(GF(2^8, 'a'), 'x', implementation="singular")
sage: f = R('a^20*x^3+x^2+a^10')
sage: f._singular_().sage_poly(R) == f
True

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: RL=R.ringlist()
[1]: 0
[2]: [1]:

sage_structured_str_list()

If self is a Singular list of lists of Singular elements, returns corresponding Sage list of lists of strings.

EXAMPLES:

sage: R=singular.ring(0, '(x,y)', 'dp')
sage: RL=R.ringlist()
sage: RL[1]:
0
[2]: [1]:

(continues on next page)
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', '\text{n}1'], ['C', '0']], '0']]

\textbf{set\_ring()}

Sets the current ring in Singular to be self.

EXAMPLES:

\begin{verbatim}
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
\end{verbatim}

\begin{verbatim}
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
\end{verbatim}

\textbf{type()}

Returns the internal type of this element.

EXAMPLES:

\begin{verbatim}
sage: R = PolynomialRing(GF(2^8,'a'),2,'x')
sage: R._singular_().type()
'ring'
sage: fs = singular('x0^2','poly')
sage: fs.type()
'poly'
\end{verbatim}
exception sage.interfaces.singular.SingularError
    Bases: RuntimeError
    Raised if Singular printed an error message

class sage.interfaces.singular.SingularFunction(parent, name)
    Bases: sage.interfaces.expect.ExpectFunction

class sage.interfaces.singular.SingularFunctionElement(obj, name)
    Bases: sage.interfaces.expect.FunctionElement

class sage.interfaces.singular.SingularGBDefaultContext(singular=None)
    Bases: object
    Within this context all Singular Groebner basis calculations are reduced automatically.

    AUTHORS:
    • Martin Albrecht
    • Simon King

class sage.interfaces.singular.SingularGBLogPrettyPrinter(verbosity=1)
    Bases: object
    A device which prints Singular Groebner basis computation logs more verbatim.

    flush()
    EXAMPLES:

    sage: from sage.interfaces.singular import SingularGBLogPrettyPrinter
    sage: s3 = SingularGBLogPrettyPrinter(verbosity=3)
    sage: s3.flush()

    write(s)
    EXAMPLES:

    sage: from sage.interfaces.singular import SingularGBLogPrettyPrinter
    sage: s3 = SingularGBLogPrettyPrinter(verbosity=3)
    sage: s3.write("(S:1337)"
    Performing complete reduction of 1337 elements.
    sage: s3.write("M[389,12]"
    Parallel reduction of 389 elements with 12 non-zero output elements.

sage.interfaces.singular.generate_docstring_dictionary()
    Generate global dictionaries which hold the docstrings for Singular functions.

    EXAMPLES:

    sage: from sage.interfaces.singular import generate_docstring_dictionary
    sage: generate_docstring_dictionary()

sage.interfaces.singular.get_docstring(name)
    Return the docstring for the function name.

    INPUT:
    • name - a Singular function name

    EXAMPLES:
sage: from sage.interfaces.singular import get_docstring
sage: 'groebner' in get_docstring('groebner')
True
sage: 'standard.lib' in get_docstring('groebner')
True

sage.interfaces.singular.is_SingularElement(x)
Returns True if x is of type SingularElement.

EXAMPLES:

sage: from sage.interfaces.singular import is_SingularElement
sage: is_SingularElement(singular(2))
True
sage: is_SingularElement(2)
False

sage.interfaces.singular.reduce_load_Singular()
EXAMPLES:

sage: from sage.interfaces.singular import reduce_load_Singular
sage: reduce_load_Singular()
Singular

sage.interfaces.singular.singular_console()
Spawn a new Singular command-line session.

EXAMPLES:

sage: singular_console()  # not tested
SINGULAR  
A Computer Algebra System for Polynomial Computations  
by: G.-M. Greuel, G. Pfister, H. Schoenemann  
FB Mathematik der Universitaet, D-67653 Kaiserslautern

sage.interfaces.singular.singular_gb_standard_options(func)
Decorator to force a reduced Singular groebner basis.

Note: This decorator is used automatically internally so the user does not need to use it manually.

sage.interfaces.singular.singular_version()
Return the version of Singular being used.

EXAMPLES:

sage: singular.version()
"Singular ... version 4..."
SYMPY --> SAGE CONVERSION

The file consists of _sage_() methods that are added lazily to the respective SymPy objects. Any call of the _sympy_() method of a symbolic expression will trigger the addition. See sage.symbolic.expression_conversion.SymPyConverter for the conversion to SymPy.

Only Function objects where the names differ need their own _sage()_ method. There are several functions with differing name that have an alias in Sage that is the same as the name in SymPy, so no explicit translation is needed for them:

```python
sage: from sympy import Symbol, Si, Ci, Shi, Chi, sign
dsage: sx = Symbol('x')
dsage: assert sin_integral(x)._sympy_() == Si(sx)
dsage: assert sin_integral(x) == Si(sx)._sage_()
dsage: assert sinh_integral(x)._sympy_() == Shi(sx)
dsage: assert sinh_integral(x) == Shi(sx)._sage_()
dsage: assert cos_integral(x)._sympy_() == Ci(sx)
dsage: assert cos_integral(x) == Ci(sx)._sage_()
dsage: assert cosh_integral(x)._sympy_() == Chi(sx)
dsage: assert cosh_integral(x) == Chi(sx)._sage_()
dsage: assert sgn(x)._sympy_() == sign(sx)
dsage: assert sgn(x) == sign(sx)._sage_()
```

AUTHORS:
- Ralf Stephan (2017-10)

```python
class sage.interfaces.sympy.UndefSageHelper
    Bases: object

    Helper class to convert sympy function objects to sage functions

    EXAMPLES:
```
```python
sage: from sympy import Function
sage: f = function('f')
sage: F = Function('f')
sage: assert f._sympy_() == F
sage: assert f == F._sage_()
```

```python
sage.interfaces.sympy.check_expression(expr, var_symbols, only_from_sympy=False)
    Does eval(expr) both in Sage and SymPy and does other checks.

    EXAMPLES:
```
sage: from sage.interfaces.sympy import check_expression
sage: check_expression("1.123*x", "x")

sage.interfaces.sympy.sympy_init(*args, **kwargs)
Add _sage_() methods to SymPy objects where needed.

This gets called with every call to Expression._sympy_() so there is only need to call it if you bypass _sympy_() to create SymPy objects. Note that SymPy objects have _sage_() methods hard installed but having them inside Sage as one file makes them easier to maintain for Sage developers.

EXAMPLES:

sage: from sage.interfaces.sympy import sympy_init
sage: from sympy import Symbol, Abs
sage: sympy_init()

sage: assert abs(x) == Abs(Symbol('x'))._sage_()

sage.interfaces.sympy.sympy_set_to_list(set, vars)
Convert all set objects that can be returned by SymPy’s solvers.

sage.interfaces.sympy.test_all()
Call some tests that were originally in SymPy.

EXAMPLES:

sage: from sage.interfaces.sympy import test_all
sage: test_all()
AUTHOR:

• John E. Stone

class sage.interfaces.tachyon.TachyonRT
    Bases: sage.structure.sage_object.SageObject

The Tachyon Ray Tracer

tachyon_rt(model, outfile='sage.png', verbose=1, block=True, extra_opts="")

INPUT:

• model - a string that describes a 3d model in the Tachyon modeling format. Type tachyon_rt.help() 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 - bool (default: True); if False, run the rendering command in the background.

• extra_opts - passed directly to tachyon command line. Use tachyon_rt.usage() to see some of the possibilities.

OUTPUT:

• Some text may be displayed onscreen.

• The file outfile is created.

EXAMPLES:
__call__ (model, outfile='sage.png', verbose=1, extra_opts='')

This executes the tachyon program, given a scene file input.

INPUT:

- model – string. The tachyon model.
- outfile – string, default 'sage.png'. The filename to save the model to.
- verbose – 0, 1, (default) or 2. The verbosity level.
- extra_opts – string (default: empty string). Extra options that will be appended to the tachyon commandline.

EXAMPLES:

```python
sage: from sage.interfaces.tachyon import TachyonRT
sage: tgen = Tachyon()
sage: tgen.texture('t1')
sage: tgen.sphere((0,0,0),1,'t1')

sage: tgen.str()[30:40]
'resolution'

sage: t = TachyonRT()

sage: import os

sage: t(tgen.str(), outfile=os.devnull)
```

help (use_pager=True)

Prints (pages) the help file written by John Stone describing scene files for Tachyon. The output is paged unless use_pager=False.

usage (use_pager=True)

Returns the basic description of using the Tachyon raytracer (simply what is returned by running tachyon with no input). The output is paged unless use_pager=False.
This module contains tools to write the .c files needed for TIDES [TIDES].

Tides is an integration engine based on the Taylor method. It is implemented as a c library. The user must translate its initial value problem (IVP) into a pair of .c files that will then be compiled and linked against the TIDES library. The resulting binary will produce the desired output. The tools in this module can be used to automate the generation of these files from the symbolic expression of the differential equation.

##########################################################################
# Copyright (C) 2014 Miguel Marco <mmarco@unizar.es>, Marcos Rodriguez
# <marcos@uunizar.es>
# # Distributed under the terms of the GNU General Public License (GPL):
# # http://www.gnu.org/licenses/
##########################################################################

AUTHORS:
• Miguel Marco (06-2014) - Implementation of tides solver
• Marcos Rodriguez (06-2014) - Implementation of tides solver
• Alberto Abad (06-2014) - tides solver
• Roberto Barrio (06-2014) - tides solver

REFERENCES:
• [ABBR2012]
• [TIDES]

sage.interfaces.tides.genfiles_mintides(integrator, driver, f, ics, initial, final, delta, tolrel=1e-16, tolabs=1e-16, output="")

Generate the needed files for the min_tides library.

INPUT:
• integrator – the name of the integrator file.
• driver – the name of the driver file.
• f – the function that determines the differential equation.
• ics – a list or tuple with the initial conditions.
• initial – the initial time for the integration.
• final – the final time for the integration.
• delta – the step of the output.
• tolrel – the relative tolerance.
• tolabs – the absolute tolerance.
• output – the name of the file that the compiled integrator will write to

This function creates two files, integrator and driver, that can be used later with the min_tides library [TIDES].

```python
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='tides')
```

Generate the needed files for the mpfr module of the tides library.

**INPUT:**
- integrator – the name of the integrator file.
- driver – the name of the driver file.
- f – the function that determines the differential equation.
- ics – a list or tuple with the initial conditions.
- initial – the initial time for the integration.
- final – the final time for the integration.
- delta – the step of the output.
- parameters – the variables inside the function that should be treated as parameters.
- parameter_values – the values of the parameters for the particular initial value problem.
- dig – the number of digits of precision that will be used in the integration
- tolrel – the relative tolerance.
- tolabs – the absolute tolerance.
- output – the name of the file that the compiled integrator will write to

This function creates two files, integrator and driver, that can be used later with the tides library ([TIDES]).

```python
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.

```python
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)])
```

```python
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.

**EXAMPLES:**

```python
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], ['mul', a, a, 'add', 1, a^2, 'sqrt', -a^2 + 1, arcsin(a)])
```
sage.interfaces.tides.subexpressions_list(f, pars=None)

Construct the lists with the intermediate steps on the evaluation of the function.

**INPUT:**

- $f$ – a symbolic function of several components.
- **pars** – a list of the parameters that appear in the function this should be the symbolic constants that appear in $f$ but are not arguments.

**OUTPUT:**

- a list of the intermediate subexpressions that appear in the evaluation of $f$.
- a list with the operations used to construct each of the subexpressions. each element of this list is a tuple, formed by a string describing the operation made, and the operands.

For the trigonometric functions, some extra expressions will be added. These extra expressions will be used later to compute their derivatives.

**EXAMPLES:**

```python
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)
([['mul', x, x], ['add', y, x^2], ['sin', x], ['cos', x], ['log', y], ['div', log(y), cos(x)])
```

```python
sage: f(a)=[cos(a), arctan(a)]
sage: from sage.interfaces.tides import subexpressions_list
sage: subexpressions_list(f)
```
sage: from sage.interfaces.tides import subexpressions_list

sage: var(‘s,b,r’)

(s, b, r)

sage: f(t,x,y,z)= [s*(y-x),x*(r-z)-y,x*y-b*z]

sage: subexpressions_list(f,[s,b,r])

([-y, x - y, s*(x - y), -s*(x - y), -z, r - z, (r - z)*x, -y, (r - z)*x - y, x*y, b*z, -b*z, x*y - b*z],
 [[‘mul’, -1, y],
  (‘add’, -y, x),
  (‘mul’, x - y, s),
  (‘mul’, -1, s*(x - y)),
  (‘mul’, -1, z),
  (‘add’, -z, r),
  (‘mul’, x, r - z),
  (‘mul’, -1, y),
  (‘add’, -y, (r - z)*x),
  (‘mul’, y, x),
  (‘mul’, z, b),
  (‘mul’, -1, b*z),
  (‘add’, -b*z, x*y])

sage: var(‘x, y’)

(x, y)

sage: f(x,y)=[exp(x^2+sin(y))]

sage: from sage.interfaces.tides import *

sage: subexpressions_list(f)

([x^2, sin(y), cos(y), x^2 + sin(y), e^(x^2 + sin(y))],
 [[‘mul’, x, x],
  (‘sin’, y),
  (‘cos’, y),
  (‘add’, sin(y), x^2),
  (‘exp’, x^2 + sin(y))])
INTERFACE TO THE SAGE CLEANER

Trivia Note: For the name “sage-cleaner”, think of the “The Cleaner” from Pulp Fiction: http://www.frankjankowski.de/quiz/illus/keitel.jpg

sage.interfaces.cleaner.cleaner(pid, cmd="")
    Write a line to the spawned_processes file with the given pid and cmd.

sage.interfaces.cleaner.start_cleaner()
    Start sage-cleaner in a new process group.
QUITTING INTERFACES

sage.interfaces.quit.expect_quitall(\texttt{verbose=False})

\textbf{EXAMPLES:}

\begin{verbatim}
sage: sage.interfaces.quit.expect_quitall()
sage: gp.eval('a=10')
'10'
sage: gp('a')
10
sage: sage.interfaces.quit.expect_quitall()
sage: gp('a')
a
sage: sage.interfaces.quit.expect_quitall(\texttt{verbose=True})
Exiting PARI/GP interpreter with PID ... running .../gp --fast --emacs --quiet --
˓
→ stacksize 1000000
\end{verbatim}

sage.interfaces.quit.invalidate_all()

Invalidate all of the expect interfaces.

This is used, e.g., by the fork-based \@parallel decorator.

\textbf{EXAMPLES:}

\begin{verbatim}
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.)
\end{verbatim}

However the maxima and gp sessions should still work out, though with their state reset:

\begin{verbatim}
sage: a = maxima(2); b = gp(3) sage: a, b (2, 3)
\end{verbatim}

sage.interfaces.quit.is_running(\texttt{pid})

Return True if and only if there is a process with id \texttt{pid} running.

sage.interfaces.quit.kill_spawned_jobs(\texttt{verbose=False})

\textbf{INPUT:}

- \texttt{verbose} – bool (default: False); if True, display a message each time a process is sent a kill signal
EXAMPLES:

```python
sage: gp.eval('a=10')
'10'
sage: sage.interfaces.quit.kill_spawned_jobs(verbse=False)
sage: sage.interfaces.quit.expect_quitall()
sage: gp.eval('a=10')
'10'
sage: sage.interfaces.quit.kill_spawned_jobs(verbse=True)
Killing spawned job ...
```

After doing the above, we do the following to avoid confusion in other doctests:

```python
sage: sage.interfaces.quit.expect_quitall()
```
AN INTERFACE TO READ DATA FILES

```
sage.interfaces.read_data.read_data(f, t)
  Read data from file ‘f’ and class ‘t’ (one element per line), and returns a list of elements.

  INPUT:
    • ‘f’ - a file name
    • ‘t’ - a class (objects will be coerced to that class)

  OUTPUT:
    a list of elements of class ‘t’.

  EXAMPLES:
```

```
sage: indata = tmp_filename()
sage: f = open(indata, "w")
sage: _ = f.write("17
42"
)
sage: f.close()
sage: l = read_data(indata, ZZ); l
[17, 42]
sage: f = open(indata, "w")
sage: _ = f.write("1.234
5.678"
)
sage: f.close()
sage: l = read_data(indata, RealField(17)); l
[1.234, 5.678]
```
INDICES AND TABLES

- Index
- Module Index
- Search Page
<table>
<thead>
<tr>
<th>Module Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage.interfaces.axiom</td>
<td>15</td>
</tr>
<tr>
<td>sage.interfaces.cleaner</td>
<td>389</td>
</tr>
<tr>
<td>sage.interfaces.ecm</td>
<td>21</td>
</tr>
<tr>
<td>sage.interfaces.expect</td>
<td>9</td>
</tr>
<tr>
<td>sage.interfaces.four_ti_2</td>
<td>27</td>
</tr>
<tr>
<td>sage.interfaces.fricas</td>
<td>33</td>
</tr>
<tr>
<td>sage.interfaces.froby</td>
<td>45</td>
</tr>
<tr>
<td>sage.interfaces.gap</td>
<td>49</td>
</tr>
<tr>
<td>sage.interfaces.gap3</td>
<td>59</td>
</tr>
<tr>
<td>sage.interfaces.gfan</td>
<td>67</td>
</tr>
<tr>
<td>sage.interfaces.giac</td>
<td>69</td>
</tr>
<tr>
<td>sage.interfaces.gnuplot</td>
<td>79</td>
</tr>
<tr>
<td>sage.interfaces.gp</td>
<td>81</td>
</tr>
<tr>
<td>sage.interfaces.interface</td>
<td>3</td>
</tr>
<tr>
<td>sage.interfaces.jmoldata</td>
<td>91</td>
</tr>
<tr>
<td>sage.interfaces.kash</td>
<td>93</td>
</tr>
<tr>
<td>sage.interfaces.kenzo</td>
<td>103</td>
</tr>
<tr>
<td>sage.interfaces.latte</td>
<td>131</td>
</tr>
<tr>
<td>sage.interfaces.lie</td>
<td>135</td>
</tr>
<tr>
<td>sage.interfaces.lisp</td>
<td>143</td>
</tr>
<tr>
<td>sage.interfaces.macaulay2</td>
<td>147</td>
</tr>
<tr>
<td>sage.interfaces.magma</td>
<td>161</td>
</tr>
<tr>
<td>sage.interfaces.magma_free</td>
<td>181</td>
</tr>
<tr>
<td>sage.interfaces.maple</td>
<td>183</td>
</tr>
<tr>
<td>sage.interfaces.mathematica</td>
<td>191</td>
</tr>
<tr>
<td>sage.interfaces.mathics</td>
<td>201</td>
</tr>
<tr>
<td>sage.interfaces.matlab</td>
<td>213</td>
</tr>
<tr>
<td>sage.interfaces.maxima</td>
<td>219</td>
</tr>
<tr>
<td>sage.interfaces.maxima_abstract</td>
<td>229</td>
</tr>
<tr>
<td>sage.interfaces.maxima_lib</td>
<td>247</td>
</tr>
<tr>
<td>sage.interfaces.mupad</td>
<td>257</td>
</tr>
<tr>
<td>sage.interfaces.mwrank</td>
<td>261</td>
</tr>
<tr>
<td>sage.interfaces.octave</td>
<td>265</td>
</tr>
<tr>
<td>sage.interfaces.phc</td>
<td>271</td>
</tr>
<tr>
<td>sage.interfaces.polymake</td>
<td>279</td>
</tr>
<tr>
<td>sage.interfaces.povray</td>
<td>295</td>
</tr>
<tr>
<td>sage.interfaces.psage</td>
<td>297</td>
</tr>
<tr>
<td>sage.interfaces.qepcad</td>
<td>299</td>
</tr>
<tr>
<td>sage.interfaces.qsieve</td>
<td>325</td>
</tr>
<tr>
<td>sage.interfaces.quit</td>
<td>391</td>
</tr>
<tr>
<td>sage.interfaces.read_data</td>
<td>393</td>
</tr>
<tr>
<td>sage.interfaces.r</td>
<td>329</td>
</tr>
<tr>
<td>sage.interfaces.rubik</td>
<td>343</td>
</tr>
<tr>
<td>sage.interfaces.sage0</td>
<td>347</td>
</tr>
<tr>
<td>sage.interfaces.sagespawn</td>
<td>13</td>
</tr>
<tr>
<td>sage.interfaces.scilab</td>
<td>353</td>
</tr>
<tr>
<td>sage.interfaces.singular</td>
<td>359</td>
</tr>
<tr>
<td>sage.interfaces.sympy</td>
<td>381</td>
</tr>
<tr>
<td>sage.interfaces.tachyon</td>
<td>383</td>
</tr>
<tr>
<td>sage.interfaces.tides</td>
<td>385</td>
</tr>
<tr>
<td>Module Index</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Symbols

__call__() (sage.interfaces.tachyon.TachyonRT method), 383
_eval_line() (sage.interfaces.polymake.PolymakeExpect method), 288
_sage_() (sage.interfaces.fricas.FriCASElement method), 39
_sage_() (sage.interfaces.macaulay2.Macaulay2Element method), 153

A
A() (sage.interfaces.qepcad.qepcad_formula_factory method), 316
after_print_text() (sage.interfaces.macaulay2.Macaulay2Element method), 155
alexander_dual() (sage.interfaces.frobby.Frobby method), 45
all_but_finitely_many() (sage.interfaces.qepcad.qepcad_formula_factory method), 318
and_() (sage.interfaces.qepcad.qepcad_formula_factory method), 318
answer() (sage.interfaces.qepcad.Qepcad method), 307
application() (sage.interfaces.polymake.PolymakeAbstract method), 280
application() (sage.interfaces.polymake.PolymakeExpect method), 290
arguments() (sage.interfaces.maxima_abstract.MaximaAbstractElementFunction method), 243
as_type() (sage.interfaces.axiom.PanAxiomElement method), 18
AsciiArtString (class in sage.interfaces.interface), 3
assign_names() (sage.interfaces.magma.MagmaElement method), 174
AssignNames() (sage.interfaces.magma.MagmaElement method), 174
associated_primes() (sage.interfaces.frobby.Frobby method), 46
assume() (sage.interfaces.qepcad.Qepcad method), 307
atomic() (sage.interfaces.qepcad.qepcad_formula_factory method), 318
Attach() (sage.interfaces.magma.Magma method), 165
attach() (sage.interfaces.magma.Magma method), 166
attach_spec() (sage.interfaces.magma.Magma method), 167
AttachSpec() (sage.interfaces.magma.Magma method), 165
attrib() (sage.interfaces.singular.SingularElement method), 372
attribute() (sage.interfaces.interface.InterfaceElement method), 5
available_packages() (sage.interfaces.r.R method), 333
Axiom (class in sage.interfaces.axiom), 17
axiom_console() (in module sage.interfaces.axiom), 19
AxiomElement (in module sage.interfaces.axiom), 17
AxiomExpectFunction (in module sage.interfaces.axiom), 17
AxiomFunctionElement (in module sage.interfaces.axiom), 17

B
bar_call() (sage.interfaces.magma.Magma method), 167
basis() (sage.interfaces.kenzo.KenzoChainComplex method), 106
BicomplexSpectralSequence() (in module sage.interfaces.kenzo), 103
blackbox() (sage.interfaces.phc.PHC method), 271
bool() (sage.interfaces.fricas.FriCASElement method), 42
build_morphism() (in module sage.interfaces.kenzo), 128

C
C() (sage.interfaces.qepcad.qepcad_formula_factory method), 316
call() (sage.interfaces.four_ti_2.FourTi2 method), 27
call() (sage.interfaces.interface.Interface method), 3
call() (sage.interfaces.r.R method), 334
cartesian_product() (sage.interfaces.kenzo.KenzoSimplicialSet method), 399
cputime() (sage.interfaces.macaulay2.Macaulay2 method), 149
cputime() (sage.interfaces.magma.Magma method), 168
cputime() (sage.interfaces.maple.Maple method), 186
cputime() (sage.interfaces.maxima_abstract.MaximaAbstract method), 230
cputime() (sage.interfaces.mupad.Mupad method), 257
cputime() (sage.interfaces.qsieve.qsieve_nonblock method), 326
cputime() (sage.interfaces.sage0.Sage method), 349
cputime() (sage.interfaces.singular.Singular method), 365
CubexSolver (class in sage.interfaces.rubik), 343
current_ring() (sage.interfaces.singular.Singular method), 365
current_ring_name() (sage.interfaces.singular.Singular method), 365
data_to_list() (in module sage.interfaces.qsieve), 325
de_solve() (sage.interfaces.maxima_abstract.MaximaAbstract method), 231
de_solve_laplace() (sage.interfaces.maxima_abstract.MaximaAbstract method), 231
de_system_plot() (sage.interfaces.octave.Octave method), 268
definition() (sage.interfaces.maxima_abstract.MaximaAbstract method), 244
degree() (sage.interfaces.kenzo.KenzoChainComplexMorphism method), 111
demo() (sage.interfaces.maxima_abstract.MaximaAbstract method), 232
derivative() (sage.interfaces.maxima_abstract.MaximaAbstract method), 238
describe() (sage.interfaces.maxima_abstract.MaximaAbstract method), 232
destructive_change_source_target_complex() (sage.interfaces.kenzo.KenzoChainComplexMorphism method), 112
detach() (sage.interfaces.expect.Expect method), 9
diff() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 239
differential() (sage.interfaces.kenzo.KenzoChainComplexMorphism method), 106
differential() (sage.interfaces.kenzo.KenzoSpectralSequence method), 123
DikSolver (class in sage.interfaces.rubik), 343
dimension() (sage.interfaces.frobby.Frobby method), 46
directory() (sage.interfaces.four_ti_2.FourTi2 method), 28
display2d() (sage.interfaces.maxima.MaximaElement method), 227
display2d() (sage.interfaces.maxima_lib.MaximaLibElement method), 250
done() (sage.interfaces.qsieve.qsieve_nonblock method), 326
dot() (sage.interfaces.macaulay2.Macaulay2Element method), 156
dot() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 239
dot_product() (sage.interfaces.r.RElement method), 340
dummy_integrate() (in module sage.interfaces.maxima_lib), 251

de_solve() (sage.interfaces.maxima_abstract.MaximaAbstract method), 231
ecl() (sage.interfaces.maxima_lib.MaximaLibElement method), 251

E
E() (sage.interfaces.gp.gp_element method), 316
ecl() (sage.interfaces.maxima_lib.MaximaLibElement method), 251
ECl() (class in sage.interfaces.ecm), 21
EilenbergMacLaneSpace() (in module sage.interfaces.kenzo), 103
eigenspace() (sage.interfaces.sage0.Sage method), 349
elements() (sage.interfaces.macaulay2.Macaulay2 method), 149
eval() (sage.interfaces.expect.Expect method), 10
eval() (sage.interfaces.fricas.Fricas method), 37
E() (sage.interfaces.gap.Gap generic method), 120
eval() (sage.interfaces.kenzo.KenzoSimplicialSet method), 120
eval() (sage.interfaces.kash.Kash method), 100
eval() (sage.interfaces.lisp.Lisp method), 144
eval() (sage.interfaces.macaulay2.Macaulay2 method), 149
eval() (sage.interfaces.magma.Magma method), 169
eval() (sage.interfaces.magma_free.MagmaElement method), 174
eval() (sage.interfaces.mathematica.Mathematica method), 197
eval() (sage.interfaces.mathics.Mathics method), 207
eval() (sage.interfaces.sage0.Sage method), 297
eval() (sage.interfaces.sage.r.Sage method), 335
eval() (sage.interfaces.sage0.Sage method), 349
evaluate() (sage.interfaces.magma.MagmaElement method), 175
evaluation() (sage.interfaces.kenzo.KenzoChainComplex method), 113
execute() (sage.interfaces.kenzo.KenzoChainComplex method), 258
frobby (class in sage.interfaces.froby), 45
full_typename() (sage.interfaces.polymake.PolymakeElement method), 284
Frobby (class in sage.interfaces.froby), 45
function() (sage.interfaces.maxima_abstract.MaximaAbstract method), 233
function_call() (sage.interfaces.gap.Gap_element method), 54
function_call() (sage.interfaces.matrix.Matrix method), 187
function() (sage.interfaces.matrix.Matrix method), 4
function() (sage.interfaces.polymake.PolymakeElement method), 100
function_call() (sage.interfaces.lie.LiE method), 140
function_call() (sage.interfaces.lisp.Lisp method), 144
function_call() (sage.interfaces.magma.Magma method), 170
function_call() (sage.interfaces.polymake.PolymakeAbstract method), 281
function_call() (sage.interfaces.r.R method), 335
FunctionElement (class in sage.interfaces.expect), 12
Gap3 (class in sage.interfaces.gap3), 43
gap_console() (in module sage.interfaces.gap3), 65
gap3_version() (in module sage.interfaces.gap3), 66
GAP3Element (class in sage.interfaces.gap3), 62
GAP3Record (class in sage.interfaces.gap3), 63
gap_command() (in module sage.interfaces.gap), 56
gap_console() (in module sage.interfaces.gap), 56
Gap_element (class in sage.interfaces.gap), 53
Gfp_element (class in sage.interfaces.gap), 53
GElement (class in sage.interfaces.gap), 53
gap_element (class in sage.interfaces.gap), 53
gap_element_generic (class in sage.interfaces.gap), 53
gap_function (class in sage.interfaces.gap), 53
gap_function_element (class in sage.interfaces.gap), 53
gc_disabled (class in sage.interfaces.expect), 12
gen() (sage.interfaces.fricas.FriCASElement method), 43
gen() (sage.interfaces.interface.InterfaceElement method), 6
gen() (sage.interfaces.magma.MagmaElement method), 175
gen_names() (sage.interfaces.magma.MagmaElement method), 176
generate_docstring_dictionary() (in module sage.interfaces.singular), 378
genfiles_mintides() (in module sage.interfaces.tides), 385
genfiles_mpfr() (in module sage.interfaces.tides), 386
gens() (sage.interfaces.magma.MagmaElement method), 176
get() (sage.interfaces.axiom.PanAxiom method), 17
get() (sage.interfaces.fricas.FriCAS method), 37
get() (sage.interfaces.gap.Gap method), 52
get() (sage.interfaces.giac.Giac method), 74
get() (sage.interfaces.gp.Gp method), 83
get() (sage.interfaces.gp.Interface method), 4
get() (sage.interfaces.kash.Kash method), 100
get() (sage.interfaces.lisp.LiE method), 140
get() (sage.interfaces.lisp.Lisp method), 144
get() (sage.interfaces.macaulay2.Macaulay2 method), 149
get() (sage.interfaces.magma.Magma method), 170
get() (sage.interfaces.maple.Maple method), 187
get() (sage.interfaces.mathematica.Mathematica method), 197
get() (sage.interfaces.mathics.Mathics method), 207
get() (sage.interfaces.matlab.Matlab method), 215
get() (sage.interfaces.maxima.Maxima method), 226
get() (sage.interfaces.maxima_lib.MaximaLib method), 249
get() (sage.interfaces.mupad.Mupad method), 258
get() (sage.interfaces.octave.Octave method), 268
get() (sage.interfaces.polymake.PolymakeAbstract method), 281
get() (sage.interfaces.psage.PSage method), 297
get() (sage.interfaces.r.R method), 335
get() (sage.interfaces.sage0.Sage method), 349
get() (sage.interfaces.scilab.Scilab method), 356
get() (sage.interfaces.singular.Singular method), 367
get_boolean() (sage.interfaces.fricas.FriCAS method), 38
get_classified_solution_dicts() (in module sage.interfaces.singular), 276
get_docstring() (sage.interfaces.gp.Gp method), 84
get_docstring() (in module sage.interfaces.singular), 378
get_InputForm() (sage.interfaces.fricas.FriCAS method), 38
get_integer() (sage.interfaces.fricas.FriCAS method), 38
get_last_params() (sage.interfaces.ecm.ECM method), 24
get_magma_attribute() (sage.interfaces.magma.MagmaElement method), 176
get_member() (sage.interfaces.polymake.PolymakeElement method), 284
get_member_function() (sage.interfaces.polymake.PolymakeElement method), 285
get_nththreads() (sage.interfaces.magma.Magma method), 171
get_precision() (sage.interfaces.gp.Gp method), 84
get_real_precision() (sage.interfaces.gp.Gp method), 84
get_record_element() (sage.interfaces.gap.Gap_method), 55
get_seed() (sage.interfaces.interface.Interface method), 4
get_series_precision() (sage.interfaces.gp.Gp method), 84
get_solution_dicts() (in module sage.interfaces.singular), 276
get_string() (sage.interfaces.fricas.FriCAS method), 38
get_unparsed_InputForm() (sage.interfaces.fricas.FriCAS method), 38
get_using_file() (sage.interfaces.interface.Interface method), 4
get_using_file() (sage.interfaces.interface.InterfaceElement method), 6
get_using_file() (sage.interfaces.lie.LiE method), 140
get_variable_list() (in module sage.interfaces.singular), 277
get_verbose() (sage.interfaces.magma.Magma method), 171
GetNthreads() (sage.interfaces.magma.Magma method), 165
GetVerbose() (sage.interfaces.magma.Magma method), 166
Gfan (class in sage.interfaces.gfan), 67
gf2_gap_to_sage() (in module sage.interfaces.gap), 56
Giac (class in sage.interfaces.giac), 72
giac_console() (in module sage.interfaces.giac), 77
GiacElement (class in sage.interfaces.giac), 75
GiacFunction (class in sage.interfaces.giac), 76
GiacFunctionElement (class in sage.interfaces.giac), 77
Gnuplot (class in sage.interfaces.gnuplot), 79
gnuplot() (sage.interfaces.gnuplot.Gnuplot method), 79
gnuplot() (in module sage.interfaces.gnuplot), 80
Gp (class in sage.interfaces.gp), 82
gp_console() (in module sage.interfaces.gp), 88
gp_version() (in module sage.interfaces.gp), 88
GpElement (class in sage.interfaces.gp), 87
graver() (sage.interfaces.four_ti_2.FourTi2 method), 28
groebner() (sage.interfaces.four_ti_2.FourTi2 method), 28
group() (sage.interfaces.kenzo.KenzoSpectralSequence method), 124

H
hasattr() (sage.interfaces.interface.InterfaceElement method), 6
help() (sage.interfaces.gap.Gap method), 52
help() (sage.interfaces.gap3.Gap3 method), 65
help() (sage.interfaces.giac.Giac method), 75
help() (sage.interfaces.gp.Gp method), 85
help() (sage.interfaces.interface.Interface method), 4
help() (sage.interfaces.interface.InterfaceFunctionElement method), 7
help() (sage.interfaces.kash.Kash method), 100
help() (sage.interfaces.lie.Lie method), 140
help() (sage.interfaces.lisp.Lisp method), 144
help() (sage.interfaces.macaulay2.Macaulay2 method), 150
help() (sage.interfaces.magma.Magma method), 171
help() (sage.interfaces.maple.Maple method), 187
help() (sage.interfaces.matheuristic.Matheuristic method), 197
help() (sage.interfaces.mathics.Mathics method), 208
help() (sage.interfaces.maxima_abstract.MaximaAbstract method), 233
help() (sage.interfaces.polymake.PolymakeAbstract method), 281
help() (sage.interfaces.r.R method), 335
help() (sage.interfaces.tachyon.TachyonRT method), 384
help_search() (sage.interfaces.kash.Kash method), 101
HelpExpression (class in sage.interfaces.r), 333
hilbert() (sage.interfaces.four_ti_2.FourTi2 method), 28
hilbert() (sage.interfaces.frobby.Frobby method), 46
homology() (sage.interfaces.kenzo.KenzoChainComplex method), 107
homotopy_group() (sage.interfaces.kenzo.KenzoSimplicialSet method), 120

I
ideal() (sage.interfaces.macaulay2.Macaulay2 method), 150
ideal() (sage.interfaces.magma.Magma method), 171
ideal() (sage.interfaces.magma.MagmaElement method), 177
ideal() (sage.interfaces.singular.Singular method), 367
identity_morphism() (sage.interfaces.kenzo.KenzoChainComplex method), 108
iff() (sage.interfaces.qepcad.qepcad_formula_factor method), 321
imag() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 239
implies() (sage.interfaces.qepcad.qepcad_formula_factor method), 321
index() (sage.interfaces.qepcad.QepcadCell method), 310
infinitely_many() (sage.interfaces.qepcad.qepcad_formula_factor method), 321
install_packages() (sage.interfaces.r.R method), 336
integral() (sage.interfaces.giac.GiacElement method), 75
integral() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 240
integral() (sage.interfaces.maxima_abstract.MaximaAbstractElementFunction method), 244
integrate() (in module sage.interfaces.latte), 132
integrate() (sage.interfaces.giac.GiacElement method), 76
integrate() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 240
integrate() (sage.interfaces.maxima_abstract.MaximaAbstractElementFunction method), 244
interact() (sage.interfaces.ecm.EM method), 24
interact() (sage.interfaces.singular.Singular method), 79
interact() (sage.interfaces.interface.Interface method), 4
Interface (class in sage.interfaces.interface), 3
InterfaceElement (class in sage.interfaces.interface), 5
InterfaceFunction (class in sage.interfaces.interface), 7
InterfaceFunctionElement (class in sage.interfaces.interface), 7
interrupt() (sage.interfaces.expect.Expect method), 10
interrupt() (sage.interfaces.gap.Gap_generic method), 55
intmod_gap_to_sage() (in module sage.interfaces.quit), 57
invalidate_all() (in module sage.interfaces.quit), 391
irreducible_decomposition() (sage.interfaces.frobby.Frobby method), 47
is_AxiomElement() (in module sage.interfaces.axiom), 19
is.ExpectElement() (in module sage.interfaces.expect), 12
is_FriCASElement() (in module sage.interfaces.quit), 391
load() (sage.interfaces.singular.Singular method), 368
load_package() (sage.interfaces.gap.Gap_generic method), 55
log() (sage.interfaces.qsieve.qsieve_nonblock method), 326
loop_space() (sage.interfaces.kenzo.KenzoSimplicialSet method), 121

M
Macaulay2 (class in sage.interfaces.macauly2), 148
Macaulay2.options() (in module sage.interfaces.macauly2), 150
macaulay2_console() (in module sage.interfaces.macauly2), 158
Macaulay2Element (class in sage.interfaces.macauly2), 152
Macaulay2Function (class in sage.interfaces.macauly2), 158
Macaulay2FunctionElement (class in sage.interfaces.macauly2), 158
Magma (class in sage.interfaces.magma), 164
magma_console() (in module sage.interfaces.magma), 180
magma_free_eval() (in module sage.interfaces.magma), 181
magma_gb_standard_options() (in module sage.interfaces.magma), 180
MagmaElement (class in sage.interfaces.magma), 174
MagmaExpr (class in sage.interfaces.magma_free), 181
MagmaFree (class in sage.interfaces.magma_free), 181
MagmaFunction (class in sage.interfaces.magma), 179
MagmaFunctionElement (class in sage.interfaces.magma), 179
MagmaGBDefaultContext (class in sage.interfaces.magma), 179
MagmaGBLogPrettyPrinter (class in sage.interfaces.magma), 179
make_cells() (sage.interfaces.qepcad.Qepcad method), 308
Maple (class in sage.interfaces.maple), 186
maple_console() (in module sage.interfaces.maple), 189
MapleElement (class in sage.interfaces.maple), 189
MapleFunction (class in sage.interfaces.maple), 189
MapleFunctionElement (class in sage.interfaces.maple), 189
Mathematica (class in sage.interfaces.mathematica), 196
mathematica_console() (in module sage.interfaces.mathematica), 198
MathematicaElement (class in sage.interfaces.mathematica), 197
MathematicaFunction (class in sage.interfaces.mathematica), 198
MathematicaFunctionElement (class in sage.interfaces.mathematica), 198
Mathics (class in sage.interfaces.mathics), 206
mathics_console() (in module sage.interfaces.mathics), 210
MathicsElement (class in sage.interfaces.mathics), 208
Matlab (class in sage.interfaces.matlab), 215
matlab_console() (in module sage.interfaces.matlab), 216
matlab_version() (in module sage.interfaces.matlab), 217
MatlabElement (class in sage.interfaces.matlab), 216
matrix() (sage.interfaces.kenzo.KenzoSimplicialSet method), 124
matrix() (sage.interfaces.singular.Singular method), 369
max_at_to_sage() (in module sage.interfaces.maxima_lib), 252
max_harmonic_to_sage() (in module sage.interfaces.maxima_lib), 252
max_to_sr() (in module sage.interfaces.maxima_lib), 253
max_to_string() (in module sage.interfaces.maxima_lib), 253
Maxima (class in sage.interfaces.maxima), 225
maxima_console() (in module sage.interfaces.maxima_abstract), 245
maxima_version() (in module sage.interfaces.maxima_abstract), 245
MaximaAbstract (class in sage.interfaces.maxima_abstract), 229
MaximaAbstractElement (class in sage.interfaces.maxima_abstract), 238
MaximaAbstractElementFunction (class in sage.interfaces.maxima_abstract), 243
MaximaElement (class in sage.interfaces.maxima), 227
MaximaElementFunction (class in sage.interfaces.maxima), 227
MaximaLib (class in sage.interfaces.maxima_lib), 248
MaximaLibElement (class in sage.interfaces.maxima_lib), 250
MaximaLibElementFunction (class in sage.interfaces.maxima_lib), 251
mdiff_to_sage() (in module sage.interfaces.maxima_lib), 253
methods() (sage.interfaces.mathematica.MagmaElement method), 177
minimize() (sage.interfaces.four_ti_2.FourTi2 method), 29
mixed_volume() (sage.interfaces.phc.PHC method), 272
mlist_to_sage() (in module sage.interfaces.mathematica_lib), 253

octave_console() (in module sage.interfaces.octave), 270
OctaveElement (class in sage.interfaces.octave), 270
one_curve() (sage.interfaces.ecm.EM, 24
op() (sage.interfaces.maple.MapleElement method), 189
operations() (sage.interfaces.gap3.GAP3Record method), 63
opposite() (sage.interfaces.kenzo.KenzoChainComplex method), 114
OptimalSolver (class in sage.interfaces.singular), 344
option() (sage.interfaces.singular.Singular method), 369
or_() (sage.interfaces.qepcad.qepcad_formula_factory method), 322
orgn() (sage.interfaces.kenzo.KenzoChainComplex method), 109
pairing() (in module sage.interfaces.kenzo), 129
PanAxiom (class in sage.interfaces.axiom), 17
PanAxiomElement (class in sage.interfaces.axiom), 18
PanAxiomExpectFunction (class in sage.interfaces.axiom), 19
PanAxiomFunctionElement (class in sage.interfaces.axiom), 19
parse_max_string() (in module sage.interfaces.maxima_lib), 254
parse_moutput_from_json() (in module sage.interfaces.mathematica), 198
partial_fraction_decomposition() (sage.interfaces.maxima_abstract.MaximaAbstract method), 242
path() (sage.interfaces.expect.Expect method), 10
path_track() (sage.interfaces.phc.PHC method), 272
phase() (sage.interfaces.qepcad.Qepcad method), 308
PHC (class in sage.interfaces.phc), 271
PHC_object (class in sage.interfaces.phc), 274
pid() (sage.interfaces.expect.Expect method), 10
pid() (sage.interfaces.qsieve.qsieve_nonblock method), 326
plot() (sage.interfaces.gnuplot.Gnuplot method), 79
plot() (sage.interfaces.r.R method), 336
plot2d() (sage.interfaces.maxima_abstract.MaximaAbstract method), 234
plot2d_parametric() (sage.interfaces.maxima_abstract.MaximaAbstract method), 234
plot3d() (sage.interfaces.gnuplot.Gnuplot method), 79
plot3d() (sage.interfaces.maxima_abstract.MaximaAbstract method), 235
plot3d_parametric() (sage.interfaces.gnuplot.Gnuplot method), 79
plot3d_parametric() (sage.interfaces.maxima_abstract.MaximaAbstract method), 234
plot_list() (sage.interfaces.maxima_abstract.MaximaAbstract method), 236
plot_multilist() (sage.interfaces.maxima_abstract.MaximaAbstract method), 236
plot_multilist() (sage.interfaces.phc.PHC method), 273
png() (sage.interfaces.r.R method), 338
Polymake (in module sage.interfaces.polymake), 279
polymake_console() (in module sage.interfaces.polymake), 293
PolymakeAbstract (class in sage.interfaces.polymake), 279
PolymakeElement (class in sage.interfaces.polymake), 283
PolymakeError, 287
PolymakeExpect (class in sage.interfaces.polymake), 287
PolymakeFunctionElement (class in sage.interfaces.polymake), 290
PolymakeJuPyMake (class in sage.interfaces.polymake), 290
POVRay (class in sage.interfaces.povray), 295
ppi() (sage.interfaces.four_ti_2.FourTi2 method), 29
preparse() (sage.interfaces.sage0.Sage method), 349
PSage (class in sage.interfaces.p sage), 297
PSageElement (class in sage. interfaces.p sage), 298
pyobject_to_max() (in module sage.interfaces.maxima_lib), 255
Q
Qepcad (class in sage.interfaces.qepcad), 307
qepcad() (in module sage.interfaces.qepcad), 313
qepcad_banner() (in module sage.interfaces.qepcad), 315
qepcad_console() (in module sage.interfaces.qepcad), 315
Qepcad_expect (class in sage.interfaces.qepcad), 312
qepcad_formula_factory (class in sage.interfaces.qepcad), 316
qepcad_version() (in module sage.interfaces.qepcad), 323
QepcadCell (class in sage.interfaces.qepcad), 310
QepcadFunction (class in sage.interfaces.qepcad), 312
qformula (class in sage.interfaces.qepcad), 323
qsieve() (in module sage.interfaces.qsieve), 325
qsieve_block() (in module sage.interfaces.qsieve), 326
qsieve_nonblock (class in sage.interfaces.qsieve), 326
qsolve() (sage.interfaces.four_ti_2.FourTi2 method), 29
qualified_typename() (sage.interfaces.polymake.PolymakeElement
quit()

quantifier()

quit()

r_console()

R

quo()

read()

rand_seed()

r_version()

recfields()

rays()

reduce_load()

reduce_load_Axiom()

reduce_load_element()

real()

read_matrix()

read_data()

recommended_B1()

reduce_load()  (in module sage.interfaces.r, 333)
r_consolid()  (in module sage.interfaces.r, 341)
r_version()  (in module sage.interfaces.r, 342)
rand_seed()  (sage.interfaces.interface.Interface method, 4)
rays()  (sage.interfaces.four_ti_2.FourTi2 method, 29)
read()  (sage.interfaces.interface.Interface method, 5)
read()  (sage.interfaces.lie.LIE method, 141)
read()  (sage.interfaces.r.R method, 338)
read_data()  (in module sage.interfaces.read_data, 393)
read_matrix()  (sage.interfaces.four_ti_2.FourTi2 method, 30)
ready()  (sage.interfaces.rubik.OptimalSolver method, 344)
real()  (sage.interfaces.maxima.abstract.MaximaAbstract method, 242)
recfields()  (sage.interfaces.gap3.GAP3Record method, 63)
recommended_B1()  (sage.interfaces.ecm.EMC method, 25)
reduce_load()  (in module sage.interfaces.mathematica, 198)
reduce_load()  (in module sage.interfaces.mathics, 211)
reduce_load_Axiom()  (in module sage.interfaces.axiom, 19)
reduce_load_element()  (in module sage.interfaces.sage0, 350)
reduce_load_fricas()  (in module sage.interfaces.fricas, 43)
reduce_load_GAP()  (in module sage.interfaces.gap, 58)
reduce_load_Giac()  (in module sage.interfaces.giac, 77)
reduce_load_GP()  (in module sage.interfaces.gp, 89)
reduce_load_Kash()  (in module sage.interfaces.kash, 101)
reduce_load_lie()  (in module sage.interfaces.lie, 142)
reduce_load_Lisp()  (in module sage.interfaces.lisp, 145)
reduce_load_macaulay2()  (in module sage.interfaces.macaulay2, 159)
reduce_load_Magma()  (in module sage.interfaces.magma, 180)
reduce_load_Maple()  (in module sage.interfaces.maple, 189)
reduce_load_Matlab()  (in module sage.interfaces.matlab, 217)
reduce_load_Maxima()  (in module sage.interfaces.maxima, 228)
reduce_load_Maxima_function()  (in module sage.interfaces.maxima_abstract, 245)
reduce_load_MaximaLib()  (in module sage.interfaces.maxima_lib, 255)
reduce_load_mupad()  (in module sage.interfaces.mupad, 258)
reduce_load_Octave()  (in module sage.interfaces.octave, 270)
reduce_load_Polymake()  (in module sage.interfaces.polymake, 294)
reduce_load_R()  (in module sage.interfaces.r, 342)
reduce_load_Sage()  (in module sage.interfaces.sage0, 350)
reduce_load_Singular()  (in module sage.interfaces.singular, 379)
RElement  (class in sage.interfaces.r, 340)
remove_constants()  (in module sage.interfaces.tides), 386
remove_output_labels()  (in module sage.interfaces.macaulay2), 159
remove_repeated()  (in module sage.interfaces.tides), 386
request_wolfram_alpha()  (in module sage.interfaces.mathematica), 198
require()  (sage.interfaces.r.R method, 338)
restart()  (sage.interfaces.macaulay2.Macaulay2 method, 151)
RFunction  (class in sage.interfaces.r), 341
RFunctionElement  (class in sage.interfaces.r), 341
ring()  (sage.interfaces.macaulay2.Macaulay2 method, 151)
r

s2k_dictmat()  (in module sage.interfaces.kenzo), 129
s2k_listofmorphisms()  (in module sage.interfaces.kenzo), 130
s2k_matrix()  (in module sage.interfaces.kenzo), 130
SAbstractSimplex()  (in module sage.interfaces.kenzo), 125
Sage  (class in sage.interfaces.sage0), 347
sage()  (sage.interfaces.interface.InterfaceElement method, 7)
sage.interfaces.axiom

module, 15
sage.interfaces.cleaner

Index
module, 389
sage.interfaces.ecm
  module, 21
sage.interfaces.expect
  module, 9
sage.interfaces.four_ti_2
  module, 27
sage.interfaces.fricas
  module, 33
sage.interfaces.froby
  module, 45
sage.interfaces.gap
  module, 49
sage.interfaces.gap3
  module, 59
sage.interfaces.gfan
  module, 67
sage.interfaces.giac
  module, 69
sage.interfaces.gmpy
  module, 79
sage.interfaces.gp
  module, 81
sage.interfaces.interface
  module, 3
sage.interfaces.jmoldata
  module, 91
sage.interfaces.kash
  module, 93
sage.interfaces.kenzo
  module, 103
sage.interfaces.latte
  module, 131
sage.interfaces.lie
  module, 135
sage.interfaces.lisp
  module, 143
sage.interfaces.macaulay2
  module, 147
sage.interfaces.magma
  module, 161
sage.interfaces.magma_free
  module, 181
sage.interfaces.maple
  module, 183
sage.interfaces.mathematica
  module, 191
sage.interfaces.mathics
  module, 201
sage.interfaces.matlab
  module, 213
sage.interfaces.maxima
  module, 219
sage.interfaces.maxima_abstract
  module, 229
sage.interfaces.maxima_lib
  module, 247
sage.interfaces.mupad
  module, 257
sage.interfaces.mwrank
  module, 267
sage.interfaces.octave
  module, 265
sage.interfaces.phc
  module, 271
sage.interfaces.polymake
  module, 279
sage.interfaces.povray
  module, 295
sage.interfaces.psage
  module, 297
sage.interfaces.qepcad
  module, 299
sage.interfaces.qsieve
  module, 325
sage.interfaces.quit
  module, 391
sage.interfaces.r
  module, 329
sage.interfaces.read_data
  module, 393
sage.interfaces.rubik
  module, 343
sage.interfaces.sage0
  module, 347
sage.interfaces.sagespawn
  module, 13
sage.interfaces.scilab
  module, 353
sage.interfaces.singular
  module, 359
sage.interfaces.sympy
  module, 381
sage.interfaces.tachyon
  module, 383
sage.interfaces.tides
  module, 385
sage0_console() (in module sage.interfaces.sage0), 350
sage0_version() (in module sage.interfaces.sage0), 350
sage2matlab_matrix_string()
  (sage.interfaces.matlab.Matlab method), 216
sage2octave_matrix_string()
  (sage.interfaces.octave.Octave method), 268
Index 413

sub() (sage.interfaces.magma.MagmaElement method), 178
subexpressions_list() (in module sage.interfaces.tides), 387
subs() (sage.interfaces.macaulay2.Macaulay2Element method), 157
subst() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 243
substitute() (sage.interfaces.macaulay2.Macaulay2Element method), 158
subtract() (sage.interfaces.kenzo.KenzoChainComplexMorphism method), 116
sum() (sage.interfaces.giac.GiacElement method), 76
sum() (sage.interfaces.kenzo.KenzoChainComplexMorphism method), 117
suspension() (sage.interfaces.kenzo.KenzoChainComplexMorphism method), 122
sw_spectral_sequence() (sage.interfaces.kenzo.KenzoSpectralSequence method), 384

T

table() (sage.interfaces.kenzo.KenzoSpectralSequence method), 125
TachyonRT (class in sage.interfaces.tachyon), 383
target_complex() (sage.interfaces.kenzo.KenzoChainComplexMorphism method), 118
temp_project() (sage.interfaces.four_ti_2.FourTi2 method), 30
tensor_product() (sage.interfaces.kenzo.KenzoChainComplexMorphism method), 109
terminate_async() (sage.interfaces.sagespawn.SagePtyProcess method), 13
test_all() (in module sage.interfaces.sympy), 382
tilde() (sage.interfaces.sagemath.RElement method), 341
time() (sage.interfaces.ecm.EMC method), 25
time() (sage.interfaces.qsieve.qsieve_nonblock method), 327
to_complex() (in module sage.interfaces.octave), 270
to_latte_polynomial() (in module sage.interfaces.latte), 133
to_poly_solve() (sage.interfaces.maxima_lib.MaximaLibElement method), 251
to_sage() (sage.interfaces.macaulay2.Macaulay2Element method), 158
type() (sage.interfaces.axiom.PanAxiomElement method), 18
type() (sage.interfaces.lie.LiEElement method), 141
type() (sage.interfaces.singular.SingularElement method), 377
typename() (sage.interfaces.polymake.PolymakeElement method), 286
typeof() (sage.interfaces.polymake.PolymakeElement method), 286

U

unapply() (sage.interfaces.giac.GiacElement method), 76
unbind() (sage.interfaces.gap.Gap_generic method), 55

UnderscoreHelper (class in sage.interfaces.sympy), 381
underscore() (sage.interfaces.macaulay2.Macaulay2Element method), 158
unit_quadratic_integer() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 237
unparsed_input_form() (sage.interfaces.axiom.PanAxiomElement method), 18
usage() (sage.interfaces.povray.POVRay method), 295
usage() (sage.interfaces.tachyon.TachyonRT method), 384
use() (sage.interfaces.macaulay2.Macaulay2 method), 152
user_dir() (sage.interfaces.sage0.Sage method), 11

V

validate_mwrank_input() (in module sage.interfaces.mwrank), 262
variable_list() (sage.interfaces.phc.PHC_Object method), 262
version() (sage.interfaces.gap.Gap_method), 56
version() (sage.interfaces.giac.Giac method), 75
version() (sage.interfaces.gp.Gp method), 87
version() (sage.interfaces.kash.Kash method), 101
version() (sage.interfaces.lie.LiE method), 141
version() (sage.interfaces.lisp.Lisp method), 145
version() (sage.interfaces.macaulay2.Macaulay2 method), 152
version() (sage.interfaces.magma.Magma method), 174
version() (sage.interfaces.matlab.Matlab method), 216
version() (sage.interfaces.maxima_abstract.MaximaAbstract method), 237
version() (sage.interfaces.octave.Octave method), 270
version() (sage.interfaces.polymake.PolymakeElement method), 286
version() (sage.interfaces.r.R method), 339
version() (sage.interfaces.sage0.Sage method), 350
version() (sage.interfaces.scilab.Scilab method), 357
version() (sage.interfaces.singular.Singular method), 372
W
wedge() (sage.interfaces.kenzo.KenzoSimplicialSet method), 123
whos() (sage.interfaces.matlab.Matlab method), 216
whos() (sage.interfaces.scilab.Scilab method), 357
with_package() (sage.interfaces.maple.Maple method), 188
write() (sage.interfaces.magma.MagmaGBLogPrettyPrinter method), 179
write() (sage.interfaces.singular.SingularGBLogPrettyPrinter method), 378
write_array() (sage.interfaces.four_ti_2.FourTi2 method), 30
write_matrix() (sage.interfaces.four_ti_2.FourTi2 method), 30
write_single_row() (sage.interfaces.four_ti_2.FourTi2 method), 31

X
X() (sage.interfaces.gp.gp.gpmod_formal_formula_factor method), 317

Z
zsolve() (sage.interfaces.four_ti_2.FourTi2 method), 31