CONTENTS

1 Common Interface Functionality .................................................. 3
2 Common Interface Functionality through Pexpect ............................ 9
3 Sage wrapper around pexpect's spawn class and ............................. 13
4 Interface to Axiom ................................................................... 15
5 The Elliptic Curve Factorization Method ....................................... 21
6 Interface to 4ti2 ....................................................................... 27
7 Interface to FriCAS .................................................................... 33
8 Interface to Froby 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 ....................................................... 50
  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
16 Interface to KASH
  16.1 Issues ................................................................. 93
  16.2 Tutorial .............................................................. 93
  16.3 Long Input .......................................................... 99

17 Interface to LattE integrale programs 101

18 Interface to LiE
  18.1 Tutorial .............................................................. 105

19 Lisp Interface 113

20 Interface to Macaulay2 117
  20.1 Parameters .......................................................... 127
  20.2 Multiple Return Values ........................................... 127
  20.3 Long Input .......................................................... 128
  20.4 Garbage Collection ................................................ 128
  20.5 Other Examples .................................................... 129

21 Interface to Magma
  21.1 Parameters .......................................................... 127
  21.2 Multiple Return Values ........................................... 127
  21.3 Long Input .......................................................... 128
  21.4 Garbage Collection ................................................ 128
  21.5 Other Examples .................................................... 129

22 Interface to the free online MAGMA calculator 147

23 Interface to Maple
  23.1 Tutorial .............................................................. 149

24 Interface to Mathematica
  24.1 Tutorial .............................................................. 158
  24.2 Long Input .......................................................... 161
  24.3 Loading and saving ................................................ 161
  24.4 Complicated translations ......................................... 161

25 Interface to MATLAB
  25.1 Tutorial .............................................................. 165

26 Pexpect interface to Maxima
  26.1 Tutorial .............................................................. 170
  26.2 Examples involving matrices ..................................... 172
  26.3 Laplace Transforms ............................................... 172
  26.4 Continued Fractions ............................................... 173
  26.5 Special examples .................................................. 173
  26.6 Miscellaneous ...................................................... 174
  26.7 Interactivity ......................................................... 174
  26.8 Latex Output ........................................................ 175
  26.9 Long Input .......................................................... 175

27 Abstract interface to Maxima 179

28 Library interface to Maxima 197

29 Interface to MuPAD 209

30 Interface to mwrank 213

31 Interface to GNU Octave
  31.1 Computation of Special Functions ............................... 217
31.2 Tutorial ................................................................. 218

32 Interface to PHC......................................................... 223

33 Interface to polymake ................................................. 231

34 POV-Ray, The Persistence of Vision Ray Tracer ......................... 243

35 Parallel Interface to the Sage interpreter .......................... 245

36 Interface to QEPCAD ................................................. 247

37 Interface to Bill Hart’s Quadratic Sieve ............................ 271

38 Interfaces to R ....................................................... 275

39 Interface to several Rubik’s cube solvers. ......................... 289

40 Interface to Sage .................................................... 291

41 Interface to Scilab .................................................... 295

42 Interface to Singular ................................................ 301
   42.1 Introduction ...................................................... 301
   42.2 Tutorial ........................................................... 301
   42.3 Computing the Genus ......................................... 304
   42.4 An Important Concept ....................................... 304
   42.5 Long Input ...................................................... 305

43 SymPy –> Sage conversion ......................................... 321

44 The Tachyon Ray Tracer ............................................. 323

45 Interface to TIDES .................................................. 325

46 Interface to the Sage cleaner ...................................... 329

47 Quitting interfaces .................................................. 331

48 An interface to read data files .................................... 333

49 Indices and Tables ................................................. 335

Python Module Index .................................................. 337

Index ........................................................................... 339
Sage provides a unified interface to the best computational software. This is accomplished using both C-libraries (see C/C++ Library Interfaces) and interpreter interfaces, which are implemented using pseudo-tty’s, system files, etc. This chapter is about these interpreter interfaces.

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

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

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

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

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

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

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

COMMON INTERFACE FUNCTIONALITY

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

AUTHORS:

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

```python
class sage.interfaces.interfaceAsciiArtString
Bases: str
```

```python
class sage.interfaces.interfaceInterface(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.

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

**function\_call** (function, args=None, kwds=None)

**EXAMPLES:**

```
sage: maxima.quad_qags(x, x, 0, 1, epsrel=1e-4)
[0.5, 0.55511151231257..., e-14, 21, 0]
sage: maxima.function_call('quad_qags', [x, x, 0, 1], {'epsrel':'1e-4'})
[0.5, 0.55511151231257..., e-14, 21, 0]
```

**get\(var\)**

Get the value of the variable var.

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

**get\_seed()**

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

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

**EXAMPLES:**

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

**get\_using\_file\(var\)**

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

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

**help\(s\)**

**interact()**

This allows you to interactively interact with the child interpreter. Press Ctrl-D or type ‘quit’ or ‘exit’ to exit and return to Sage.

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

**name\(new\_name=None\)**

**new\(code\)**

**rand\_seed()**

Return a random seed that can be put into set\_seed function for any interpreter.

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

```python
sage: from sage.interfaces.interface import Interface
sage: i = Interface("")
sage: i.rand_seed() # random
318491487L
sage: s = Singular()
sage: s.rand_seed() # random
365260051L
```

**read (filename)**

EXAMPLES:

```python
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)
```

**set (var, value)**

Set the variable var to the given value.

**set_seed (seed=None)**

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

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

If seed is None then should generate a random seed.

EXAMPLES:

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

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

```python
class sage.interfaces.interface.InterfaceElement (parent, value, is_name=False, name=None)
Bases: sage.structure.element.Element

Interface element.

attribute (attrname)

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

EXAMPLES:
```
sage: 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 ] ]

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

bool()
Return whether this element is equal to True.

NOTE:
This method needs to be overridden if the subprocess would not return a string representation of a boolean value unless an explicit print command is used.

EXAMPLES:

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

\texttt{gen(n)}
get\_using\_file()
Return this element's string representation using a file. Use this if self has a huge string representation. It'll be way faster.

EXAMPLES:

```
sage: a = maxima(str(2^1000))
sage: a.get\_using\_file()
˓→'10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983...
˓→'
```

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

EXAMPLES:

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

\texttt{is\_string()}  
Tell whether this element is a string.

By default, the answer is negative.

\texttt{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
sage: x = r([1,2,3]); x
[1] 1 2 3
sage: x.name()
'sage...'
sage: x = r([1,2,3]).name('x'); x
[1] 1 2 3
sage: x.name()
'x'
```

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

```python
class sage.interfaces.interface.InterfaceFunction(parent, name)
    Bases: sage.structure.sage_object.SageObject

Interface function.

class sage.interfaces.interface.InterfaceFunctionElement(obj, name)
    Bases: sage.structure.sage_object.SageObject

Interface function element.

help()
```

```python
sage.interfaces.interface.is_InterfaceElement(x)
```
COMMON INTERFACE FUNCTIONALITY THROUGH PEXPECT

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

AUTHORS:

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

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

Bases: sage.interfaces.interface.Interface

Expect interface object.

clear_prompts ()

command ()

Returns the command used in this interface.

EXAMPLES:

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

detach ()

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

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

Calling detach() a second time does nothing:

sage: maxima.detach()

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

• code – text to evaluate

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

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

• allow_use_file – bool (default: True); if True and code exceeds an interface-specific threshold then code will be communicated via a temporary file rather 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:
```python
sage: pid = gap.pid()
sage: gap.eval('quit;')
''
sage: pid == gap.pid()
True
sage: gap.quit()
sage: pid == gap.pid()
False
```

```python
def quit(verbos=False)
    """Quit the running subprocess.
    INPUT:
    • verbose – (boolean, default False) print a message when quitting this process?
    EXAMPLES:
    sage: a = maxima('y')
sage: maxima.quit(verbos=True)
    Exiting Maxima with PID ... running .../bin/maxima ...
    sage: a._check_valid()
    Traceback (most recent call last):
      ...
    ValueError: The maxima session in which this object was defined is no longer running.
    Calling quit() a second time does nothing:
    sage: maxima.quit(verbos=True)
```

```python
def server()()
    """Returns the server used in this interface.
    EXAMPLES:
    sage: magma.set_server_and_command(server = 'remote')
    No remote temporary directory (option server_tmpdir) specified, using /tmp/ on remote
    sage: magma.server() # indirect doctest
    'remote'
```

```python
def set_server_and_command(server=None, command=None, server_tmpdir=None, ulimit=None)
    """Changes the server and the command to use for this interface. This raises a Runtime error if the interface is already started.
    EXAMPLES:
    sage: magma.set_server_and_command(server = 'remote', command = 'mymagma') # indirect doctest
    No remote temporary directory (option server_tmpdir) specified, using /tmp/ on remote
    sage: magma.server()
    'remote'
    sage: magma.command()
    """sage-native-execute ssh -t remote 'mymagma'"
```

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

sage.interfaces.expect.console(cmd)

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

    EXAMPLES:

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

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

```python
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:

```python
sage: s.ptyproc.terminate_async(interval=0.2)
sage: while True:
....:     try:
....:         os.kill(s.pid, 0)
....:     except OSError:
....:         sleep(0.1)
```

(continues on next page)
....:  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'
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.

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

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

\[
\text{sage: } a \text{.type()}
\]

PositiveInteger

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

\[
\text{sage: } F = \text{axiom.factor('x^5 - y^5'); } F
\]

\[- (y - x)(y + x y + x y + x y + x)\]

\[
\text{sage: type(F)}
\]

Factored Polynomial Integer

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

\[
\text{sage: } a = \text{axiom}(2/3); a
\]

\[\frac{2}{3}\]

\[
\text{sage: } a = \text{axiom}('x^2 + 3/7'); a
\]

\[x^2 + \frac{3}{7}\]

The \texttt{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.

\[
\text{sage: print(axiom.eval('factor(x^5 - y^5)'))}
\]

\[- (y - x)(y + x y + x y + x y + x)\]

Type: Factored Polynomial Integer

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

\[
\text{sage: } f = \text{axiom}('x^5 - y^5')
\]

\[
\text{sage: } f^2
\]

\[- 2x y + x\]

\[
\text{sage: } f \text{.factor()}
\]

\[- (y - x)(y + x y + x y + x y + x)\]

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

\[
\text{sage: } f = \text{axiom}('x^5 - y^5')^{10000}
\]

Interrupting Axiom...

\[
\text{<type 'exceptions.TypeError'>: Ctrl-c pressed while running Axiom}\]
sage: axiom('1/100 + 1/101')  #optional - axiom
201
-----
10100
sage: a = axiom('(1 + sqrt(2))^5'); a  #optional - axiom
++
29\sqrt{2} + 41
sage: a = axiom(x+2); a  #optional - axiom
x + 2
sage: a.subst(x=3)  #optional - axiom
5

We verify that Axiom floating point numbers can be converted to Python floats.

sage: float(axiom(2))  #optional - axiom
2.0

class sage.interfaces.axiom.Axiom(name='axiom', command='axiom -nox -noclef', script_subdirectory=None, logfile=None, server=None, server_tmpdir=None, init_code=[')lisp (si::readline-off)'])
Bases: 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, server_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'
sage: a = axiom('(1 + sqrt(2))^5') #optional - axiom
sage: axiom.get(a.name())  #optional - axiom
' +-+\r\n 29\|2 + 41'
```

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

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

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

as_type (type)
Returns self as type.

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

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

EXAMPLES:
```
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:
```
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:
```
sage: a = axiom(x^2+1); a        #optional - axiom
_2
```
(continues on next page)
x + l
sage: a.unparsed_input_form() #optional - axiom
'x*x+1'

class sage.interfaces.axiom.PanAxiomExpectFunction(parent, name)
    Bases: sage.interfaces.expect.ExpectFunction

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

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

    EXAMPLES:

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

sage.interfaces.axiom.is_AxiomElement(x)
    Returns True of 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
THE ELLIPTIC CURVE FACTORIZATION METHOD

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

Sage includes GMP-ECM, which is a highly optimized implementation of Lenstra's elliptic curve factorization method. See 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.

```python
class sage.interfaces.ecm.ECM(B1=10, B2=None, **kwds):
    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
- pml – 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
• `reduc` – 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

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

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

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

```python
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:

```python
sage: ecm.factor((2^197 + 1)/3)  # long time
[197002597249, 1348959352853811313, 251951573867253012259144101843]
sage: ecm.get_last_params()
# random output
{'poly': 'x^1', 'sigma': '1785694449', 'B1': '8885', 'B2': '1002846'}
```

#### interact()

Interactively interact with the ECM program.

**EXAMPLES:**

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

#### one_curve\((n,\text{factor_digits}=\text{None}, \text{B1}=2000, \text{algorithm}=\text{ECM'}, **\text{kwds})\)

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
- \( \text{factor_digits} \) – integer. Decimal digits estimate of the wanted factor.
- \( \text{B1} \) – integer. Stage 1 bound (default 2000)
- \( \text{algorithm} \) – either “ECM” (default), “P-1” or “P+1”

**OUTPUT:**

a list \([p, q]\) where \( p \) and \( q \) are integers and \( n = p \times q \). If no factor was found, then \( p = 1 \) and \( q = n \).
Warning: Neither p nor q in the output is guaranteed to be prime.

EXAMPLES:

```sage
f = ECM()
sage: n = 508021860739623467191080372196682785441177798407961
sage: f.one_curve(n, B1=10000, sigma=11)
[1, 508021860739623467191080372196682785441177798407961]
sage: f.one_curve(n, B1=10000, sigma=1022170541)
[79792266297612017, 6366805760909027985741435139224233]
sage: n = 43213288788390310800980214331445113500016816977037257
sage: f.one_curve(n, B1=500000, algorithm="P-1")
[67872792749091946529, 6366805760909027985741435139224233]
sage: n = 2088352670731726262548647919416588631875815083
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
sage: n = next_prime(11^23)*next_prime(11^37)
sage: ecm.time(n, 35) # random output
Expected curves: 910, Expected time: 23.95m
```

(continues on next page)
Input number is
\[304481639541418099574459465485462199861257489887231115912293\] (63 digits)
Using MODMULN [mulredc:0, sqrredc:0]
Using B1=250000, B2=128992510, polynomial Dickson(3), sigma=3244548117
dF=2048, k=3, d=19110, d2=11, i0=3
Expected number of curves to find a factor of n digits:
35 40 45 50 55 60 65 70 75 80
4911 70940 1226976 2.5e+07 5.8e+08 1.6e+10 2.7e+13 4e+18 5.4e+23 Inf
Step 1 took 230ms
Using 10 small primes for NTT
Estimated memory usage: 4040K
Initializing tables of differences for F took 0ms
Computing roots of F took 9ms
Building F from its roots took 16ms
Computing 1/F took 9ms
Initializing table of differences for G took 0ms
Computing roots of G took 8ms
Building G from its roots took 16ms
Computing roots of G took 7ms
Building G from its roots took 16ms
Computing G \cdot H took 6ms
Reducing G \cdot H mod F took 5ms
Computing roots of G took 7ms
Building G from its roots took 17ms
Computing G \cdot H took 5ms
Reducing G \cdot H mod F took 5ms
Computing polyeval(F,G) took 34ms
Computing product of all F(g_i) took 0ms
Step 2 took 164ms
Expected time to find a factor of n digits:
35 40 45 50 55 60 65 70 75 80
32.25m 7.76h 5.60d 114.21d 7.27y 196.42y 337811y 5e+10y 7e+15y Inf
Expected curves: 4911, Expected time: 32.25m
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=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.

    **EXAMPLES:**

    ```python
    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
    [-5 0 2]
    [-5 3 0]
    ```

    **circuits** (mat=None, project=None)
    Run the 4ti2 program **circuits** 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.circuits([[1,2,3]]) # optional - 4ti2
    [ 0  3 -2]  
    ```
```


**directory()**

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

**EXAMPLES:**

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

**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:**

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

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

**INPUT:**
- `filename` – The name of the file to read from.
OUTPUT:

The data from the file as a matrix over \( \mathbb{Z} \).

EXAMPLES:

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

```python
temp_project()
```

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

EXAMPLES:

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

```python
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 `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")
```

```python
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.
- `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_matrix([[1,2],[3,4]], "test_file")
```

```python
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.
- `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_single_row([1,2,3], "test_file")
```
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:

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

```python
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:

```python
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:

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

FriCAS objects are normally displayed using “ASCII art”:

```python
sage: fricas(2/3) #optional - fricas
2
```

(continues on next page)
Functions defined in FriCAS are available as methods of the `fricas` object:

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

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

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

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

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

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

```sage
sage: f = fricas('(x^5 - y^5)^10000')
#not tested - 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:

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

(continues on next page)
Sage Reference Manual: Interpreter Interfaces, Release 8.7

(continued from previous page)

\[ n \]

\[ [x \ f(x): \]

\[ 2 \ 3 (iv) \ 2 \ 2 , \ 3 , , , \]  

\[ x \ f(x) f(x) + (20 \ x \ f(x) f(x) + 5 \ x \ f(x) \ f(x) \]  

\[ + \ 2 \ 2 , , 2 \]  

\[ - 39 \ x \ f(x) f(x) \]  

\[ + \ 2 , 2 , 3 , 2 , 4 \]  

\[ (12 \ x \ f(x) f(x) - 15 \ x \ f(x) f(x) + 4 \ f(x) \ f(x) f(x) + 6 \ f(x) \ f(x) \]  

\[ + \ 10 \ x \ f(x) f(x) f(x) - 16 \ f(x) f(x) f(x) \]  

\[ = \ 0 \]  

\[ , \ 2 , 3 , 4 \]  

\[ f(x) = 1 + x + 2 x + 3 x + O(x) \]  

FriCAS can solve linear ordinary differential equations:

```sage
sage: fricas.set("y", "operator y")  #
˓→ optional - fricas
sage: fricas.set("deq", "x^3*D(y x, x, 3) + x^2*D(y x, x, 2) - 2*x*D(y x, x) + 2*y x - 2*x^4")  # optional - fricas
sage: fricas.set("sol", "solve(deq, y, x)"); fricas("sol")  #
˓→ optional - fricas
      5 3 2
\[ x - 10 x + 20 x + 4 \]

[particular = ----------------------,  
 \[ 15 x \]

\[ \]  

\[ 3 2 3 3 2 \]  

\[ 2 x - 3 x + 1 x - 1 x - 3 x - 1 \]  

basis = [---------------------, --------, -----------------]  
\[ x \]

sage: fricas("sol.particular").sage()  #
˓→ optional - fricas
\[ 1/15*(x^5 - 10*x^3 + 20*x^2 + 4)/x \]

sage: fricas("sol.basis").sage()  #
˓→ optional - fricas
\[ [(2*x^3 - 3*x^2 + 1)/x, (x^3 - 1)/x, (x^3 - 3*x^2 - 1)/x] \]

sage: fricas.eval("clear values y deq sol")  #
˓→ optional - fricas
''

FriCAS can expand expressions into series:

```sage
sage: x = var('x'); ex = sqrt(cos(x)); a = fricas(ex).series(x=0); a  #
˓→ optional - fricas
\[ 1/15*(x^5 - 10*x^3 + 20*x^2 + 4)/x \]

sage: fricas("sol.basis").sage()  #
˓→ optional - fricas
\[ [(2*x^3 - 3*x^2 + 1)/x, (x^3 - 1)/x, (x^3 - 3*x^2 - 1)/x] \]

sage: fricas.eval("clear values y deq sol")  #
˓→ optional - fricas
''

(continues on next page)
FriCAS does some limits right:

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

```python
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: 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:**

```python
def eval_code(code, strip=True, synchronize=False, locals=None, allow_use_file=True, split_lines='nofile', reformat=True, **kwds):
    """
    Evaluate code using FriCAS.
    """
    if strip:
        code = code.strip()
    # other code
```

**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:**

```python
def get_value(var):
    """
    Get the string representation of the value.
    """
    return var.eval()
```

---

37
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

'2'

class sage.interfaces.fricas.FriCASElement
Bases: sage.interfaces.expect.ExpectElement

Instances of this class represent objects in FriCAS.

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

```sage
fricas(2.1234).sage()
```

# optional - fricas

2.12340000000000

```sage
_.parent()
```

# optional - fricas

(continues on next page)
Real Field with 53 bits of precision

\begin{verbatim}
sage: a = RealField(100)(pi)  #
      optional - fricas
sage: fricas(a).sage()  #
      optional - fricas
3.1415926535897932384626433833
\end{verbatim}

\begin{verbatim}
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}

\begin{verbatim}
Algebraic numbers:

\begin{verbatim}
sage: a = fricas('(1 + sqrt(2))^5'); a  
      optional - fricas
+-+
29 \|2 + 41
\end{verbatim}

\begin{verbatim}
sage: b = a.sage(); b  #
      optional - fricas
82.0121933088198?
\end{verbatim}

\begin{verbatim}
sage: b.radical_expression()  #
      optional - fricas
29*sqrt(2) + 41
\end{verbatim}
\end{verbatim}

\begin{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}
\end{verbatim}

\begin{verbatim}
Matrices over a prime field:

\begin{verbatim}
sage: fricas("matrix [[1::PF 3, 2],[2, 0]]").sage().parent()  #,
      optional - fricas
Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 3
\end{verbatim}
\end{verbatim}

\begin{verbatim}
We can also convert FriCAS's polynomials to Sage polynomials:

\begin{verbatim}
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
\end{verbatim}
\end{verbatim}
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).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: fricas("(y^2+sqrt 3)::UP(y, AN)").sage()  # optional - fricas
y^2 + 1.732050807568878?

Rational functions:

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

Expressions:

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

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

sage: fricas("factorial(n)").sage()  # optional - fricas
factorial(n)

sage: fricas("integrate(sin(x+y), x=0..1)").sage()  # optional - fricas
(continues on next page)
\[ -\cos(y + 1) + \cos(y) \]

```python
sage: fricas("\texttt{integrate(x*sin(1/x), x=0..1)}")\cdot\texttt{sage()}
\texttt{\#  #}
\texttt{optional - fricas}
\texttt{\"failed\"
```

```
sage: fricas("\texttt{integrate(sin((x^2+1)/x),x)}")\cdot\texttt{sage()}
\texttt{\#  #}
\texttt{optional - fricas}
\texttt{integral(sin(x^2 + 1)/x), x}
```

**Todo:**

- Converting matrices and lists takes much too long.

**Matrices:**

```python
sage: fricas("\texttt{matrix \{[x^n/2^m for n in 0..5] for m in 0..3}\}}")\cdot\texttt{sage()}
\texttt{\# optional - fricas, long time}
\texttt{[ 1 \ x \ x^2 \ x^3 \ x^4 \ x^5 \]}
\texttt{[ 1/2 \ 1/2*x \ 1/2*x^2 \ 1/2*x^3 \ 1/2*x^4 \ 1/2*x^5 \]}
\texttt{[ 1/4 \ 1/4*x \ 1/4*x^2 \ 1/4*x^3 \ 1/4*x^4 \ 1/4*x^5 \]}
\texttt{[ 1/8 \ 1/8*x \ 1/8*x^2 \ 1/8*x^3 \ 1/8*x^4 \ 1/8*x^5 \]}
```

**Lists:**

```python
sage: fricas("\texttt{[2^n/x^n for n in 0..5]}")\cdot\texttt{sage()}
\texttt{\#  #}
\texttt{optional - fricas, long time}
\texttt{[1, 2/x, 4/x^2, 8/x^3, 16/x^4, 32/x^5]}
```

```python
sage: fricas("\texttt{[matrix \{[i for i in 1..n]\}} for n in 0..5]"\).\cdot\texttt{sage()}
\texttt{\#  #}
\texttt{optional - fricas, long time}
\texttt{[[], \[1\], \[1 2\], \[1 2 3\], \[1 2 3 4\], \[1 2 3 4 5\]}
```

**Error handling:**

```python
sage: s = fricas.guessPade("\texttt{[fibonacci i for i in 0..10]}\); s\texttt{\#  #}
\texttt{optional - fricas}
\texttt{n \ x}
\texttt{[[\[x \]- \--------\]]}
\texttt{2}
\texttt{x \ + \ x \ - \ 1}
```

```
sage: s.sage()
\texttt{\#  #}
\texttt{optional - fricas}
\texttt{Traceback (most recent call last):
...
\texttt{NotImplementedError: the translation of the FriCAS Expression \texttt{\textquoteright\texttt{rootOfADE\textquoteright}} to \texttt{sage} is not yet implemented
```
\[
\begin{align*}
- \quad x & - \quad x + O(x) \\
65536 & \quad 262144
\end{align*}
\]

```python
sage: s.sage()

# optional - fricas
Traceback (most recent call last):
...
NotImplementedError: the translation of the FriCAS object

\[
\begin{align*}
1 & \quad 1 \quad 2 \quad 1 \quad 3 \quad 5 \quad 4 \quad 7 \quad 5 \quad 21 \quad 6 \quad 33 \quad 7 \quad 429 \quad 8 \\
1 + \quad - \quad x & - \quad x + \quad \cdot \quad x & - \quad \cdot \quad x + \quad \cdot \quad x & - \quad \cdot \quad x + \quad \cdot \quad x & - \quad \cdot \quad x - \quad \cdot \quad x
\end{align*}
\]

\[
\begin{align*}
2 & \quad 8 \quad 16 \quad 128 \quad 256 \quad 1024 \quad 2048 \quad 32768 \\
+ & \quad 715 \quad 9 \quad 2431 \quad 10 \quad 11 \\
\end{align*}
\]

----- 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:**

```python
sage: fricas("1=1").bool()  # optional - fricas
True
sage: fricas("1\neq1").bool()  # optional - fricas
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:**

```python
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.
```

(continues on next page)
sage.interfaces.fricas.is_FriCASElement(x)

Return True if x is of type FriCASElement.

EXAMPLES:

```python
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()

Returns the FriCAS interface object defined in :sage.interfaces.fricas.

EXAMPLES:

```python
sage: from sage.interfaces.fricas import reduce_load_fricas
# optional - fricas
sage: reduce_load_fricas()
# optional - fricas
FriCAS
```
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

#### 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.

```python
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 \((x_1 - 1)(x_2 - 1)\ldots(x_n - 1)\) in the denominator, assuming the variables of the ring are \(x_1, x_2, \ldots, x_n\). 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.
irreducible_decomposition (monomial_ideal)

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

**INPUT:**

- monomial_ideal – The monomial ideal to decompose.

**OUTPUT:**

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

**EXAMPLES:**

This is a simple example of computing irreducible decomposition.

```
sage: (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: 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).

```
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
sage: allOK # optional - frobby
True
```

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

```
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
sage: allOK # 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 ]
```

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.

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

In order to eval $g$ in GAP, we need to tell GAP to view the variables $x_0$ and $x_1$ 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().

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

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

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.

```python
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.,

```python
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?

```python
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

console()

Spawn a new GAP command-line session.

EXAMPLES:

```python
sage: gap.console()  # not tested
********** GAP, Version 4.5.7 of 14-Dec-2012 (free software, GPL)
* GAP * http://www.gap-system.org
********** Architecture: x86_64-unknown-linux-gnu-gcc-default64
Libs used: gmp, readline
Loading the library and packages ...
Packages: GAPDoc 1.5.1
Try '?help' for help. See also '?copyright' and '?authors'
gap>
```
cputime \( (t=\text{None}) \)
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:

```
sage: t = gap.cputime()
sage: t  #random
0.13600000000000001
sage: gap.Order(gap.SymmetricGroup(5))
120
sage: gap.cputime(t)  #random
0.05999999999999998
```

get \( (\text{var, use_file=False}) \)
Get the string representation of the variable \text{var}.

EXAMPLES:

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

help \( (s, \text{pager=True}) \)
Print help on a given topic.

EXAMPLES:

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

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

save_workspace()
Save the GAP workspace.

set \( (\text{var, value}) \)
Set the variable \text{var} to the given value.

EXAMPLES:

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

set_seed \( (\text{seed=\text{None}}) \)
Set the seed for GAP interpreter.

The seed should be an integer.

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

class sage.interfaces.gap.GapElement (parent, value, is_name=False, name=None)
Bases: sage.interfaces.gap.GapElement_generic
str (use_file=False)

EXAMPLES:

sage: print(gap(2))
2

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

bool ()

EXAMPLES:

sage: bool(gap(2))
True
sage: gap(0).bool()
False
sage: gap('false').bool()
False

is_string ()
Tell whether this element is a string.

EXAMPLES:

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

class sage.interfaces.gap.GapFunction (parent, name)
Bases: sage.interfaces.expect.ExpectFunction

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:

\begin{verbatim}
sage: gap.eval('2+2') '4'
sage: gap.eval('Print(4); #test\n Print(6);') '46'
sage: gap.eval('Print("#"); Print(6);') '#6'
sage: gap.eval('4; \n 6;') '4\n6'
sage: gap.eval('if 3>2 then\nPrint("hi");\nfi;') 'hi'
sage: gap.eval('## this is a test\nPrint("OK");') 'OK'
sage: gap.eval('Print("This is a test. Oh no, a ";# but this is a 
comment\nPrint("OK")') 'This is a test. Oh no, a #OK'
sage: gap.eval('if 4>3 then'
  ''
sage: gap.eval('Print("Hi how are you?")') 'Hi how are you?'
sage: gap.eval('fi') ;;
\end{verbatim}

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

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

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

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

### get_record_element (record, name)

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

**INPUT:**
- `record` – a GAP record
- `name` – str

**OUTPUT:**
- `GapElement`

**EXAMPLES:**

```python
t = gap('rec( a := 1, b := "2" )')
gap.get_record_element(t, 'a')
1
gap.get_record_element(t, '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
gap._eval_line('while(1=1) do i:=1;; od;', wait_for_prompt=False)
''
gap.interrupt(timeout=1)
[ 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:

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

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

```
sage.interfaces.gap.gap_console()
```

Spawn a new GAP command-line session.

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

EXAMPLES:

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

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

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

The first time you start GAP from Sage, it saves the startup state of GAP in a file $HOME/.sage/gap/workspace-gap-HASH, where HASH is a hash of the directory where Sage is installed.

This is useful, since then subsequent startup of GAP is at least 10 times as fast. Unfortunately, if you install any new code for GAP, it won’t be noticed unless you explicitly load it, e.g., with gap.load_package(“my_package”)

The packages sonata, guava, factint, gapdoc, grape, grape, design, toric, and laguna are loaded in all cases before the workspace is saved, if they are available.

```
sage.interfaces.gap.get_gap_memory_pool_size()
```

Get the gap memory pool size for new GAP processes.

EXAMPLES:

```python
sage: from sage.interfaces.gap import get_gap_memory_pool_size
sage: get_gap_memory_pool_size()  # random output
1534059315
```
sage.interfaces.gap.gfq_gap_to_sage(x, F)

INPUT:

• x – GAP finite field element
• F – Sage finite field

OUTPUT: element of F

EXAMPLES:

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

AUTHOR:

• David Joyner and William Stein

sage.interfaces.gap.intmod_gap_to_sage(x)

INPUT:

• x – Gap integer mod ring element

EXAMPLES:

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

(continues on next page)
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

sage.interfaces.gap.set_gap_memory_pool_size(size_in_bytes)
Set the desired gap memory pool size.

Subsequently started GAP instances will use this as default. Already running instances are unchanged.

GAP will only reserve size_in_bytes address space. Unless you actually start a big GAP computation, the memory will not be used. However, corresponding swap space will be reserved so that GAP will always be able to use the reserved address space if needed. While nothing is actually written to disc as long as you don’t run a big GAP computation, the reserved swap space will not be available for other processes.

INPUT:


EXAMPLES:

sage: from sage.interfaces.gap import ...
    set_gap_memory_pool_size
sage: n = get_gap_memory_pool_size()
sage: set_gap_memory_pool_size(n)
sage: n == get_gap_memory_pool_size()
True
sage: n  # random output
1534059315
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')
```
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:

   ```python
   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) )
   sage: type(S5)            #optional - gap3
   <class 'sage.interfaces.gap3.GAP3Record'>
   ```

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

   ```python
   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)`:  

   ```python
   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`:

   ```python
   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.

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

   (continues on next page)
3. You can start a new GAP3 session as follows:

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

```
Lehrstuhl D fuer Mathematik
RWTH Aachen

Version 3
Release 4.4
18 Apr 97

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

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

```python
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:

```python
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):

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

Load a GAP3 package:

```
sage: gap3.load_package("chevie")  #optional - gap3
sage: gap3.version()  # random #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
Traceback (most recent call last):
...
TypeError: Gap3 produced error output
Error, Variable: 'x' must have a value
...
```

```python
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', ...]
```

recfields()

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

OUTPUT:
• list of strings - the field records

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

class sage.interfaces.gap3.Gap3(command='gap3')
Bases: sage.interfaces.gap.Gap_generic

A simple Expect interface to GAP3.

EXAMPLES:

AUTHORS:

• Franco Saliola (Feb 2010)

calendar()

    Spawn a new GAP3 command-line session.

EXAMPLES:

    sage: gap3.calendar()

    #not tested

    64 Chapter 10. Interface to GAP3
Sage Reference Manual: Interpreter Interfaces, Release 8.7

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

**help** *(topic, pager=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 sectio...
help system. The help system lets you read the manual inter...

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

```
asage.interfaces.gap3.gap3_console()  #not tested
```

Spawn a new GAP3 command-line session.

EXAMPLES:

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

10.5. Examples 65
sage.interfaces.gap3.gap3_version()

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

EXAMPLES:

```
sage: gap3_version()                    # random, optional
   gap3
   'lib: v3r4p4 1997/04/18, src: v3r4p0 1994/07/10, sys: usg gcc ansi'
```
INTERFACE TO GROEBNER FAN

AUTHOR:

• Anders Nedergaard Jensen: Write gfan C++ program, which implements algorithms many of which were in-
vented 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 \( \mathbb{Z}/p\mathbb{Z} \) for \( p \leq 32749 \)

• -- 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>

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

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

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

EXAMPLES:

```
sage: giac('3 * 5')
15
sage: giac.eval('ifactor(2005)')
'5*401'
sage: giac.ifactor(2005)
2005
sage: l=giac.ifactors(2005) ; l; l[2]
[5,1,401,1]
401
sage: giac.fsolve('x^2=cos(x)+4', 'x','0..5')
[1.9140206190...

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

```

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

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

We can write that in Sage as

```plaintext
sage: giac('factor(x^5-1)')
(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 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

```plaintext
sage: giac('(x^5-1)').factor()
(x-1)*(x^4+x^3+x^2+x+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.

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

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,

```plaintext
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)
```

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

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

Incidentally you can always get into a giac console by the command.

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

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:

```python
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.

    ```python
    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])
    ```

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

    ```python
    sage: l.factors()
    [y+sqrt(2),4]
    sage: g=giac('assume(y>0)'); giac('y^2=3').solve('y')
    y...
    ```

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

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

    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');a,giac('a'),giac(a),giac(a).evalf()
30
(sqrt(2), 5, sqrt(2), 1.41421356237309504880168872421)
```

clear(var)

Clear the variable named var.

EXAMPLES:

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

completions(s)

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

EXAMPLES:

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

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
0>>
```

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)
```

(continues on next page)
**eval** *(code, strip=True, **kwds)*

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

**INPUT:**
- code – str
- strip – Default is True and removes \n
**EXAMPLES:**

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

sage: giac(s)
{(x)->[x+1,x+2]}

sage: giac.g(5)
7
```
Sage Reference Manual: Interpreter Interfaces, Release 8.7

```python
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,\rightarrow 0)=Psi(a))...
```

**set** *(var, value)*

Set the variable var to the given value.

**EXAMPLES:**

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

**version()**

Wrapper for giac’s version().

**EXAMPLES:**

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

```python
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:**

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

**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

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

**EXAMPLES:**

```python
```
\texttt{sage:} y=\texttt{giac('y');} f=(\texttt{sin(2*y)/y).integral(y).simplify();} f
\texttt{Si(2*y)}
\texttt{sage:} f.\texttt{diff(y).simplify()}
\texttt{sin(2*y)/y}
\texttt{sage:} f = \texttt{giac('exp(x^2)').integral('x',0,1)} ; f
\texttt{1.46265174...}
\texttt{sage:} x,y=\texttt{giac('x'),giac('y');}\texttt{integrate(cos(x+y),'x=0..pi').simplify()}
\texttt{-2*sin(y)}

\textbf{sum}(\textit{var, min=None, max=None})

Return the sum of self with respect to the variable \textit{x}.

\textbf{INPUT:}

\begin{itemize}
\item \textit{var} - variable
\item \textit{min} - default: None
\item \textit{max} - default: None
\end{itemize}

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

\textbf{EXAMPLES:}

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

\textbf{unapply}(\textit{var})

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

\textbf{EXAMPLES:}

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

\texttt{class} \texttt{sage.interfaces.giac.GiacFunction}(\textit{parent, name})
\texttt{Bases:} \texttt{sage.interfaces.expect.ExpectFunction}

\texttt{class} \texttt{sage.interfaces.giac.GiacFunctionElement}(\textit{obj, name})
\texttt{Bases:} \texttt{sage.interfaces.expect.FunctionElement}

\texttt{sage.interfaces.giac.giac\_console()}

Spawn a new Giac command-line session.

\textbf{EXAMPLES:}

\texttt{sage:} \texttt{giac.console()} \# not tested - giac
\texttt{...}

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:

```
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:
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()
Type \texttt{gp[tab]} for a list of all the functions available from your Gp install. Type \texttt{gp[tab]?} for Gp’s help about a given function. Type \texttt{gp(\ldots)} to create a new Gp object, and \texttt{gp.eval(\ldots)} 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):

\begin{verbatim}
sage: M = gp('[1,2;3,4]')
sage: M
[1, 2; 3, 4]
sage: M * M
[7, 10; 15, 22]
sage: M + M
[2, 4; 6, 8]
sage: M.matdet()
-2

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

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

sage: print(gp("taylor(sin(x),x)"))
x - 1/6*x^3 + 1/120*x^5 - 1/5040*x^7 + 1/362880*x^9 - 1/39916800*x^11 + 1/
   6227020800*x^13 - 1/1307674368000*x^15 + O(x^16)
\end{verbatim}

GP has a powerful very efficient algorithm for numerical computation of integrals.

\begin{verbatim}
sage: gp("a = intnum(x=0,6,sin(x))")
0.03982971334963397945434770208  # 32-bit
0.039829713349633979454347702077075594548  # 64-bit
sage: gp("a")
0.03982971334963397945434770208  # 32-bit
0.039829713349633979454347702077075594548  # 64-bit
sage: gp.kill("a")
sage: gp("a")
a
\end{verbatim}
Note that gp ASCII plots do work in Sage, as follows:

```
sage: print(gp.eval("plot(x=0,6,sin(x))"))
0.9988963 |''''''''''''_x...x_''''''''''''''''''''''''''''''''''''''''''|
| x" "x |
| _" " |
|_" " |
|_ _ |
```

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

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

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

```
sage: f = gp('x^9 - x - 2')
sage: f.polgalois()
[362880, -1, 34, "S9"]
```

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:**

```sage
sage: Gp()
PARI/GP interpreter
```

**console()**

Spawn a new GP command-line session.

**EXAMPLES:**

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

**cputime**(t=None)

cputime for pari - cputime since the pari process was started.

**INPUT:**

- `t` - (default: None); if not None, then returns time since `t`

**Warning:** If you call `gettime` explicitly, e.g., `gp.eval('gettime')`, you will throw off this clock.

**EXAMPLES:**

```sage
sage: gp.cputime() # random output
0.00800000000000000002
sage: gp.factor('2^157-1')
[852133201, 1; 6072644167, 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:

```python
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:**

```python
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
```

**get_precision** *(*)

Return the current PARI precision for real number computations.

**EXAMPLES:**

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

**get_real_precision** *(*)

Return the current PARI precision for real number computations.

**EXAMPLES:**

```python
sage: gp.get_real_precision()
28  # 32-bit
38  # 64-bit
```

**get_series_precision** *(*)

Return the current PARI power series precision.

**EXAMPLES:**

```python
sage: gp.get_series_precision()
16
```

**help** *(command)*

Returns GP’s help for command.

**EXAMPLES:**
\texttt{sage: gp.help('gcd')}
\texttt{gcd(x,y): greatest common divisor of x and y.}'

**kill** \texttt{(var)}
Kill the value of the GP variable \texttt{var}.

**INPUT:**
- \texttt{var} (string) – a valid GP variable identifier

**EXAMPLES:**
\begin{verbatim}
sage: gp.set('xx', '22')
sage: gp.get('xx')
'22'
sage: gp.kill('xx')
sage: gp.get('xx')
'xx'
\end{verbatim}

**new\_with\_bits\_prec** \texttt{(s, precision=0)}
Creates a GP object from \texttt{s} with \texttt{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:**
\begin{verbatim}
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
3.1415926535897932384626433832795028841971693993510582098 # 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
\end{verbatim}

**set** \texttt{(var, value)}
Set the GP variable \texttt{var} to the given value.

**INPUT:**
- \texttt{var} (string) – a valid GP variable identifier
- \texttt{value} – a value for the variable

**EXAMPLES:**
### 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_real_precision(53); old_prec
28 # 32-bit
38 # 64-bit
sage: gp.get_real_precision()
57
sage: gp.set_real_precision(old_prec)
57
sage: gp.get_real_precision()
28 # 32-bit
38 # 64-bit
```
57
\begin{verbatim}
sage: gp.set_precision(old_prec)
sage: gp.get_precision()
28    # 32-bit
38    # 64-bit
\end{verbatim}

**set_seed** *(seed=None)*

Set the seed for gp interpreter.

The seed should be an integer.

**EXAMPLES:**

\begin{verbatim}
sage: g = Gp()
sage: g.set_seed(1)
1
sage: [g.random() for i in range(5)]
[1546275796, 879788114, 1745191708, 771966234, 1247963869]
\end{verbatim}

**set_series_precision** *(prec=None)*

Sets the PARI power series precision, and returns the old precision.

**EXAMPLES:**

\begin{verbatim}
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
\end{verbatim}

**version** *

Returns the version of GP being used.

**EXAMPLES:**

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

**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.

\begin{verbatim}
sage: a = gp(39393)
sage: loads(a.dumps()) == a
True
\end{verbatim}

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:

\begin{verbatim}
sage: E = gp('ellinit([1,2,3,4,5])')
sage: loads(dumps(E)) == E
True
sage: x = gp.Pi()/3
\end{verbatim}
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.

bool()

EXAMPLES:
sage: gp(2).bool()
True
sage: bool(gp(2))
True
sage: bool(gp(0))
False

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))
True
sage: is_GpElement(2)
False
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(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 (21.17 26.72 27.295) 27.544636 (0.0 0.0 0.0) -25.287832 64.8414 0.0 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")

sage: D.export_jmol(archive_name)
# not scaled properly...need some more steps.

sage: archive_native = archive_name

sage: import sys

sage: if sys.platform == 'cygwin':
....: import cygwin
....: archive_native = cygwin.cygpath(archive_native, 'w')

sage: script = 'set defaultdirectory "\nscript"'.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 install the appropriate optional Sage package by typing something like “sage -i kash3-linux-2005.11.22” or “sage -i kash3_osx-2005.11.22”. For a list of optional packages type “sage -optional”. If you type one of the above commands, the (about 16MB) package will be downloaded automatically (you don’t have to do that).

It is not enough to just have KASH installed on your computer. Note that the KASH Sage package is currently only available for Linux and OSX. If you need Windows, support contact me (wstein@gmail.com).

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 the optional kash package be installed.

#### 16.2.1 Basics

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

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

Next we obtain the result as a new KASH object.
We can do arithmetic and call functions on KASH objects:

```
sage: a*a
    484
sage: a.Factorial()
    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
    2469/5
sage: a**b
    193765903041146393565116739165642262657761441158615231767486923346401992277143215887218137603759765625
```

### 16.2.4 Variable assignment

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

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

```
sage: a.name()
    # somewhat random; optional - kash
'sage0'
sage: kash(a.name())
    # optional -- kash
32233
```

### 16.2.5 Integers and Rationals

We illustrate arithmetic with integers and rationals in KASH.
Sage Reference Manual: Interpreter Interfaces, Release 8.7

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.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
sage: z = kash('1 + 2*I')  # optional -- kash
sage: z  # 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
(continues on next page)
sage: kash.Precision(30)                           # optional -- kash
30
sage: kash.Log('3+I')                              # optional -- kash
1.60943791244100376075933332 + 0.927295218001612232428512462922*I
sage: kash.Log('I')                                # optional -- kash
1.5707963267948966192312169164*I

sage: kash.Sqrt(4)                                 # optional -- kash
2.000000000000000000000000000000
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
1.000000000000000000000000000000

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
SUCCESS
sage: v.Append_([5, 6])                           # optional -- kash
SUCCESS
sage: v                                            # optional -- kash
SUCCESS
sage: v                                            # optional -- kash

The apply command applies a function to each element of a list.

:: sage: L = kash([1,2,3,4]) # optional -- kash sage: L.Apply('i -> 3*i') # optional -- kash [ 3, 6, 9, 12 ] sage: L #

### 16.2.8 Ranges

The following are examples of ranges.

```python
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 ]
```

### 16.2.9 Sequences

### 16.2.10 Tuples

### 16.2.11 Polynomials

```python
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.Evalute(10)          # optional -- kash
1011
sage: Qx = kash.PolynomialAlgebra('Q') # optional -- kash
sage: Qx.gen(1)**5 + kash('7/3') # sage1 below somewhat random; optional -- kash
sage: Qx()**5 + 7/3
```

### 16.2.12 Number Fields

We create an equation order.

```python
sage: f = kash('X^5 + 4*X^4 - 56*X^2 - 16*X + 192') # optional -- kash
sage: OK = f.EquationOrder()                        # optional -- kash
sage: a = OK.gen(2)                                # optional -- kash
sage: a                                            # optional -- kash
[0, 1, 0, 0, 0]
sage: OK.Basis()                                  # output somewhat random; optional -- kash
[                                          
  _NG.1,
  _NG.2,
  _NG.3,
  _NG.4,
  _NG.5
```

(continues on next page)
Determining whether an ideal is principal.

```python
sage: I.IsPrincipal()
FALSE, extended by:
```

Computation of class groups and unit groups:

```python
sage: f = kash('X^5 + 4*X^4 - 56*X^2 -16*X + 192')
sage: O = kash.EquationOrder(f)
sage: OK = O.MaximalOrder()
sage: OK.ClassGroup()
Abelian Group isomorphic to Z/6
Defined on 1 generator
Relations:
6*sage32.1 = 0, extended by:
  ext1 := Mapping from: grp^abl: sage32 to ids/ord^num: _AA
```

```python
sage: U = OK.UnitGroup()
sage: U
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())
[ [1, -1, 0, 0, 0], [1, 1, 0, 0, 0], [-1, 0, 0, 0, 0] ]
```
16.2.13 Function Fields

<table>
<thead>
<tr>
<th>sage:</th>
<th>k = kash.FiniteField(25)</th>
<th># optional -- kash</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage:</td>
<td>kT = k.RationalFunctionField()</td>
<td># optional -- kash</td>
</tr>
<tr>
<td>sage:</td>
<td>kTy = kT.PolynomialAlgebra()</td>
<td># optional -- kash</td>
</tr>
<tr>
<td>sage:</td>
<td>T = kT.gen(1)</td>
<td># optional -- kash</td>
</tr>
<tr>
<td>sage:</td>
<td>y = kTy.gen(1)</td>
<td># optional -- kash</td>
</tr>
<tr>
<td>sage:</td>
<td>f = y<strong>3 + T</strong>4 + 1</td>
<td># optional -- kash</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>sage:</th>
<th>a = kash(range(10000))</th>
<th># optional -- kash</th>
</tr>
</thead>
</table>

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

cronsole()  

`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

g`et(var)`
Get the value of the variable `var`.

`help(name=None)`
Return help on KASH commands.

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') # optional -- kash

There is one entry in X for each item found in the documentation for this function: If you type
print(X[0]) you will get help on about the first one, printed nicely to the screen.

AUTHORS:

• Sebastion Pauli (2006-02-04): during Sage coding sprint

help_search(name)

set(var, value)

Set the variable var to the given value.

version()

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

Call to the program count from LattE integrale

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

Counting integer points from either the H or V representation:

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

Ehrhart polynomial:

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

Multivariate generating function currently only work with raw_output=True:

```
sage: opts = {'cdd': True,
.....:     'multivariate_generating_function': True,
.....:     'raw_output': True}
sage: cddin = P.cdd_Hrepresentation()
```
Call to the function \texttt{integrate} from LattE integrale.

\textbf{INPUT:}

- \texttt{arg} -- a cdd or LattE description string.
- \texttt{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.
- \texttt{algorithm} -- (default: 'triangulate') the integration method. Use 'triangulate' for polytope triangulation or 'cone-decompose' for tangent cone decomposition method.
- \texttt{raw_output} -- if True then return directly the output string from LattE.
- \texttt{verbose} -- if True then return directly verbose output from LattE.
- For all other options of the integrate program, consult the LattE manual.

\textbf{OUTPUT:}

Either a string (if \texttt{raw_output} if set to True) or a rational.

\textbf{EXAMPLES:}

\begin{verbatim}
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:

\begin{verbatim}
sage: integrate(P.cdd_Hrepresentation(), x^2*y^2*z^2, cdd=True)  # optional - latte_int
4096/27
sage: integrate(P.cdd_Vrepresentation(), x^2*y^2*z^2, cdd=True)  # optional - latte_int
4096/27
\end{verbatim}

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

\begin{verbatim}
sage: integrate(P.cdd_Hrepresentation(), cdd=True)  # optional - latte_int
64
sage: integrate(P.cdd_Vrepresentation(), cdd=True)  # optional - latte_int
64
\end{verbatim}

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

To access the LiE interpreter directly, run \texttt{lie}\_\texttt{console}().

\textbf{EXAMPLES:}

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

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

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

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

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

\textbf{18.1 Tutorial}

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

You can perform basic arithmetic operations in LiE.
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
sage: v # optional - lie
[3,2,6873,-38]
6873
sage: v+v # optional - lie
[6,4,13746,-76]
sage: v*v # optional - lie
47239586
sage: v+234786 # optional - lie
[3,2,6873,-38,234786]
sage: v^v # optional - lie
[3,2,6873,-38,3,2,6873,-38]
```

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]]

sage: m^3 # optional - lie
[[ 220, 87, 81, 375],
 [-168,-1089, 13,1013],
 [1550, 357,-55,1593],
 [-854, -652, 98,-170]]

sage: v*m # optional - lie
[-6960,62055,55061,-319]
sage: m*v # optional - lie
```

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

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

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

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

You can define your own functions in LiE using `lie.eval`. Once you've defined a function (say `f`), you can call it using `lie.f`; however, user-defined functions do not show up when using tab-completion.
LiE’s help can be accessed through `lie.help('functionname')` where functionname is the function you want to receive help for.

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

**Integers:**

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

**Vectors:**

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

**Matrices:**

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

**Polynomials:**

```
sage: a = lie('X[1,2] - 2*X[2,1]') # optional - lie
sage: b = a.sage(); b # optional - lie
-2*x0^2*x1 + x0*x1^2
```

(continues on next page)
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]')  # optional - lie
sage: b2 = lie('[2,3]')  # optional - lie
sage: eval_pol(a, b1)  # optional - lie
4
sage: eval_pol(a, b2)  # optional - lie
18
```

AUTHORS:

- Mike Hansen 2007-08-27
- William Stein (template)
**eval** (code, strip=True, **kwds)

EXAMPLES:

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

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

EXAMPLES:

```python
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:

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

**get_using_file** (var)

EXAMPLES:

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

**help** (command)

Returns a string of the LiE help for command.

EXAMPLES:

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

**read** (filename)

EXAMPLES:

```python
sage: filename = tmp_filename()
sage: f = open(filename, 'w')
sage: _ = f.write('x = 2
sage: f.close()
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

```
version()
```
EXAMPLES:

```
sage: lie.version()  # optional - lie
'2.1'
```

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

**type()**
EXAMPLES:

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

**class** `sage.interfaces.lie.LiEFElement` (parent, name)
**Bases**: `sage.interfaces.expect.ExpectFunction`

**class** `sage.interfaces.lie.LiEFElement` (obj, name)
**Bases**: `sage.interfaces.expect.FunctionElement`

**sage.interfaces.lie.is_LiEElement** (x)
EXAMPLES:

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

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

EXAMPLES:

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

**sage.interfaces.lie.lie_version**()

EXAMPLES:

```
sage: from sage.interfaces.lie import lie_version
sage: lie_version()  # optional - lie
'2.1'
```

**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: 10.factorial()
3628800
sage: a = lisp(17)
sage: a.factorial()
355687428096000
```


```python
class sage.interfaces.lisp.Lisp(maxread=None, script_subdirectory=None, logfile=None, server=None, server_tmpdir=None):
    Bases: sage.interfaces.expect.Expect

EXAMPLES:
```
```
**console()**

Spawn a new Lisp command-line session.

**EXAMPLES:**

```python
code: lisp.console() #not tested
```

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

EXAMPLES:

```python
code: 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:**

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

**get**(var)**

EXAMPLES:

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

**help**(command)**

EXAMPLES:

```python
code: lisp.help('setq')
Traceback (most recent call last):
...\nNotImplementedError
```

**kill**(var)**

EXAMPLES:

```python
code: lisp.kill('x')
Traceback (most recent call last):
...\nNotImplementedError
```

**set**(var, value)**

Set the variable var to the given value.

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

version()
Returns the version of Lisp being used.

EXAMPLES:

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

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

sage: lisp(2).bool()
True
sage: lisp(0).bool()
False
sage: bool(lisp(2))
True

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

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

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

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

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

EXAMPLES:

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

sage.interfaces.lisp.reduce_load_Lisp()
EXAMPLES:
sage: from sage.interfaces.lisp import reduce_load_Lisp
sage: reduce_load_Lisp()
Lisp Interpreter
Note: You must have Macaulay2 installed on your computer for this interface to work. Macaulay2 is not included with Sage, but you can obtain it from <http://www.math.uiuc.edu/Macaulay2/>. Note additional optional Sage packages are required.

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

The Macaulay2 interface offers three pieces of functionality:

- `Macaulay2_console()` – A function that dumps you into an interactive command-line Macaulay2 session.
- `Macaulay2(expr)` – Evaluation of arbitrary Macaulay2 expressions, with the result returned as a string.
- `Macaulay2.new(expr)` – Creation of a Sage object that wraps a Macaulay2 object. This provides a Pythonic interface to Macaulay2. For example, if `f=Macaulay2.new(10)`, then `f.gcd(25)` returns the GCD of 10 and 25 computed using Macaulay2.

EXAMPLES:

```python
sage: print(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[y,z]')  # optional - macaulay2
sage: print(R)  # optional - macaulay2
ZZ
--[y..z, Degrees => {3:1}, Heft => {1}, MonomialOrder => {MonomialSize => 32},
  →DegreeRank => 1]

sage: x = macaulay2('x')  # optional - macaulay2
sage: y = macaulay2('y')  # optional - macaulay2
sage: print((x+y)^5)  # optional - macaulay2
5 5
x + y

sage: parent((x+y)^5)  # optional - macaulay2
Macaulay2
```

(continues on next page)
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.

class sage.interfaces.macaulay2.Macaulay2 (maxread=None, script_subdirectory=None, logfile=None, server=None, server_tmpdir=None)

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

Interface to the Macaulay2 interpreter.

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

EXAMPLES:

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

cputime (t=None)
EXAMPLES:

```
sage: R = macaulay2("QQ[x,y]") # optional - macaulay2
sage: x,y = R.gens() # optional - macaulay2
sage: a = (x+y+1)^20 # optional - macaulay2
sage: cputime() # optional - macaulay2; random
0.483937000000001
```

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.

EXAMPLES:

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

help (s)

EXAMPLES:

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

ideal (*gens*)

Return the ideal generated by gens.

INPUT:

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

OUTPUT:

the Macaulay2 ideal generated by the given list of gens

EXAMPLES:

```
  sage: R2 = macaulay2.ring('QQ', '[x, y]'); R2
  # optional - macaulay2
  QQ[x..y, Degrees => {2:1}, Heft => {1}, MonomialOrder => {MonomialSize => 16},
  DegreeRank => 1]
    {Lex => 2 }
    {Position => Up }
  sage: I = macaulay2.ideal( ('y^2 - x^3', 'x - y') ); I
  # optional - macaulay2
  ideal (- x + y , x - y)
  sage: J = I^3; J.gb().gens().transpose()
  # optional - macaulay2
  { -9 } | y9-3y8+3y7-y6 |
  { -7 } | xy6-2xy5+xy4-y7+2y6-y5 |
  { -5 } | x2y3-x2y2-2xy4+2xy3+y5-y4 |
  { -3 } | x3-3x2y+3xy2-y3 |
```
new_from (type, value)

Return a new Macaulay2Element of type type constructed from value.

EXAMPLES:

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

restart()

Restart Macaulay2 interpreter.

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

Create a Macaulay2 ring.

INPUT:

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

OUTPUT:

- a Macaulay2 ring (with base ring ZZ)

EXAMPLES:

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

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

This is a polynomial ring over the rational numbers:

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

set (var, value)

Set the variable var to the given value.

EXAMPLES:

```python
sage: macaulay2.set("a", "2")  # optional - macaulay2
sage: macaulay2.get("a")       # optional - macaulay2
2
```
\textbf{use}(R)

Use the Macaulay2 ring \(R\).

\textbf{EXAMPLES:}

\begin{verbatim}
sage: R = macaulay2("QQ[x,y]") # optional - macaulay2
sage: P = macaulay2("ZZ/7[symbol x, symbol y]") # optional - macaulay2
sage: macaulay2("x").cls() # optional - macaulay2
ZZ
--[x..y, Degrees => {2:1}, Heft => {1}, MonomialOrder => {MonomialSize => 32},
   DegreeRank => 1]
7
  {GRevLex => {2:1} }
  {Position => Up }
sage: macaulay2.use(R) # optional - macaulay2
sage: macaulay2("x").cls() # optional - macaulay2
QQ[x..y, Degrees => {2:1}, Heft => {1}, MonomialOrder => {MonomialSize => 32},
   DegreeRank => 1]
  {GRevLex => {2:1} }
  {Position => Up }
\end{verbatim}

\textbf{version}()

Returns the version of Macaulay2.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: macaulay2.version() # optional - macaulay2
(1, 1...
\end{verbatim}

\textbf{class} \texttt{sage.interfaces.macaulay2.Macaulay2Element}(parent, value, \texttt{is_name=False}, \texttt{name=None})

\textbf{Bases:} \texttt{sage.interfaces.tab_completion.ExtraTabCompletion}, \texttt{sage.interfaces.expect.ExpectElement}

\textbf{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.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: macaulay2(ZZ).cls() # optional - macaulay2
Ring
\end{verbatim}

\textbf{dot}(x)

\textbf{EXAMPLES:}

\begin{verbatim}
sage: d = macaulay2.new("MutableHashTable") # optional - macaulay2
sage: d["k"] = 4 # optional - macaulay2
sage: d.dot("k") # optional - macaulay2
4
\end{verbatim}

\textbf{external_string}()

\textbf{EXAMPLES:}

\begin{verbatim}
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]}')
\end{verbatim}

\textbf{repr}()

\textbf{EXAMPLES:}
sage: R = macaulay2("QQ[x,y,z]/(x^3-y^3-z^3)")  # optional - macaulay2
sage: x = macaulay2('x')  # optional - macaulay2
sage: y = macaulay2('y')  # optional - macaulay2
sage: print(x+y)  # optional - macaulay2
x + y
sage: print(macaulay2("QQ[x,y,z]"))  # optional - macaulay2
QQ[x..z, Degrees => {3:1}, Heft => {1}, MonomialOrder => {MonomialSize => 32}, DegreeRank => 1]
  (GRevLex => {3:1} )
  (Position => Up  )
  QQ[x, y, z]
  --------
  x + y + z

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: f = macaulay2('x^3 + 3*y^11 + 5')  # optional - macaulay2
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()
sage: print(X.structure_sheaf())  # optional - macaulay2
OO
sage...

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[\text{x}]")  # optional - macaulay2
sage: P = macaulay2("ZZ/7[\text{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.to_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[\text{x}]")  # optional - macaulay2
sage: P = macaulay2("ZZ/7[\text{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.to_sage().parent()  # optional - macaulay2
Univariate Polynomial Ring in x over Finite Field of size 7
```

```
to_sage()

EXAMPLES:

```python
sage: macaulay2(ZZ).to_sage()  # optional - macaulay2
Integer Ring
sage: macaulay2(QQ).to_sage()  # optional - macaulay2
Rational Field
sage: macaulay2(2).to_sage()  # optional - macaulay2
2
sage: macaulay2(1/2).to_sage()  # optional - macaulay2
1/2
sage: _.parent()  # optional - macaulay2
Rational Field
sage: macaulay2([1,2,3]).to_sage()  # optional - macaulay2
[1, 2, 3]
```

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

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

(continues on next page)
Finite Field of size 7
\[ \text{sage: macaulay2(GF(49,'a')).to_sage()} \]  # optional - macaulay2
Finite Field in a of size 7^2
\[ \text{sage: R.<x,y> = QQ[]} \]
\[ \text{sage: macaulay2(x^2+y^2+1).to_sage()} \]  # optional - macaulay2
\[ x^2 + y^2 + 1 \]
\[ \text{sage: I = macaulay2("ideal (x,y)")} \]  # optional - macaulay2
\[ \text{Ideal (x, y) of Multivariate Polynomial Ring in x, y over Rational Field} \]
\[ \text{sage: R = macaulay2("QQ[x,y]")} \]  # optional - macaulay2
\[ \text{Multivariate Polynomial Ring in x, y over Rational Field} \]
\[ \text{sage: R/I} \]  # optional - macaulay2
\[ \text{Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x, y)} \]
\[ \text{sage: R = macaulay2("QQ^2")} \]  # optional - macaulay2
\[ \text{Vector space of dimension 2 over Rational Field} \]
\[ \text{sage: m = macaulay2("hello")} \]  # optional - macaulay2
\[ 'hello' \]

\text{underscore}(x)

\text{EXAMPLES:}
\[ \text{sage: a = macaulay2([1,2,3])} \]  # optional - macaulay2
\[ \text{sage: a.underscore(0)} \]  # optional - macaulay2
\[ 1 \]

\text{class} \ sage.interfaces.macaulay2.Macaulay2Function(\text{parent, name})
\text{Bases:} \ sage.interfaces.expect.ExpectFunction
\text{sage.interfaces.macaulay2.is_Macaulay2Element}(x)

\text{EXAMPLES:}
\[ \text{sage: from sage.interfaces.macaulay2 import is_Macaulay2Element} \]
\[ \text{sage: is_Macaulay2Element(2)} \]  # optional - macaulay2
False
\[ \text{sage: is_Macaulay2Element(macaulay2(2))} \]  # optional - macaulay2
True

\text{sage.interfaces.macaulay2.macaulay2_console}()

\text{Spawn a new M2 command-line session.}

\text{EXAMPLES:}
\[ \text{sage: macaulay2_console()} \]  # not tested
Macaulay 2, version 1.1

(continues on next page)
sage.interfaces.macaulay2.reduce_load_macaulay2()
Used for reconstructing a copy of the Macaulay2 interpreter from a pickle.

EXAMPLES:

```python
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:

```python
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]

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/.

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`.

### 21.1 Parameters

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

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

### 21.2 Multiple Return Values

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

```
sage: n = magma(100)                           # optional - magma
sage: n.IsSquare(nvals = 1)                     # optional - magma
true
```

(continues on next page)
We verify that an obviously principal ideal is principal:

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

### 21.3 Long Input

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

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

### 21.4 Garbage Collection

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

First, create a matrix `m` in Sage:

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

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

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

It is called `_sage_[…]` in Magma:

```
sage: s = A.name(); s
'_sage_[…]'
```

It’s there:

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

Now I delete the reference to that matrix:

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

Now `_sage_[…]` is “zeroed out” in the Magma session:
Sage Reference Manual: Interpreter Interfaces, Release 8.7

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

21.5 Other Examples

We compute a space of modular forms with character.

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

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

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

We coerce some polynomial rings into Magma:

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

(continues on next page)
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 - m
sage: y * 5
# optional - m
5*x
sage: y * 1.0
# optional - m
$.1
sage: y * (2/3)
# optional - m
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(script_subdirectory=None, logfile=None, server=None, server_tmpdir=None, user_config=False, seed=None, command=None)

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

Interface to the Magma interpreter.

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

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

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

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

```python
sage: magma.SetDefaultRealFieldPrecision(200, nvals=0)  # magma >= v2.12;
   # optional - magma
sage: magma.eval('1.1')  # optional - magma
   '1.
   →100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
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:

```python
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...
```

`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:**

```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)
```

`GetNthreads()`

Get the number of threads used in Magma.

**EXAMPLES:**

21.5. Other Examples
GetVerbose \( \text{(type)} \)
Get the verbosity level of a given algorithm class etc. in Magma.

**INPUT:**

- type - string (e.g. ‘Groebner’), see Magma documentation

**EXAMPLES:**

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

SetNthreads \( \text{(\textit{n})} \)
Set the number of threads used for parallelized algorithms in Magma.

**INPUT:**

- \textit{n} - number of threads

**EXAMPLES:**

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

SetVerbose \( \text{(\textit{type}, \textit{level})} \)
Set the verbosity level for a given algorithm, class, etc. in Magma.

**INPUT:**

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

**EXAMPLES:**

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

attach \( \text{(\textit{filename})} \)
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:
attach_spec(filename)

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

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

INPUT:

- filename - a string

EXAMPLES:

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

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

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

INPUT:

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

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

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

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

**INPUT:**
- dir \(\) – a string

**EXAMPLES:**

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

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

**INPUT:**
- var \(\) – a string

**EXAMPLES:**

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

```python
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
```

**cputime** \((\text{t} = \text{None})\)
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 \( \text{cputime since } t \)

**OUTPUT:**

- float - seconds

**EXAMPLES:**

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

**eval** (\( x, \text{strip=True, **kwds} \))

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

**INPUT:**

- \( x \) - string of code
- \( \text{strip} \) - ignored

**OUTPUT:** string

**EXAMPLES:**

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

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

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

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

Verify that trac ticket #9705 is fixed:

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

Verify that trac ticket #11401 is fixed:

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

(continues on next page)
function_call (function, args=[], params={}, nvals=1)
Return result of evaluating a Magma function with given input, parameters, and asking for nvals as output.

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

OUTPUT: MagmaElement or tuple of nvals MagmaElement’s

EXAMPLES:

sage: magma.function_call('Factorization', 100) # optional - magma
[[<2, 2>, <5, 2>]]

sage: magma.function_call('NextPrime', 100, {'Proof':False}) # optional - magma
101

sage: magma.function_call('PolynomialRing', [QQ,2]) # optional - magma
Polynomial ring of rank 2 over Rational Field
Order: Lexicographical
Variables: $.1, $.2

Next, we illustrate multiple return values:

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

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

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

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

EXAMPLES:

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

Get the number of threads used in Magma.

**EXAMPLES:**

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

**get_verbose(type)**

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

**INPUT:**

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

**EXAMPLES:**

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

**help(s)**

Return Magma help on string s.

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

**INPUT:**

- *s* - string

**OUTPUT:** string

**EXAMPLES:**

```
sage: magma.help("NextPrime")  # optional - magma
```

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

**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:
 (continues on next page)
\[
\begin{align*}
x^2, \\
x \cdot y^3
\end{align*}
\]

**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:**

```python
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;
    ....:     _ = f.write("print "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:**

```python
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:

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

**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
[13, 55, 84, 100, 37]
```

### `set_verbose(type, level)`
Set the verbosity level for a given algorithm, class, etc. in Magma.

**INPUT:**

- `type` – string (e.g. ‘Groebner’)
- `level` – integer >= 0

**EXAMPLES:**

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

### `version()`
Return the version of Magma that you have in your PATH on your computer.

**OUTPUT:**

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

**EXAMPLES:**
```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
sage: S.1 # 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
sage: S.1 # optional - magma
a
sage: S.1^2 + S.2 # optional - magma
a^2 + b
```

**eval** *(args)*

Evaluate self at the inputs.

**INPUT:**

- **args** – import arguments

**OUTPUT:** `self(*args)`

**EXAMPLES:**

```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
```

**evaluate** *(args)*

Evaluate self at the inputs.

**INPUT:**

- **args** – import arguments

**OUTPUT:** `self(*args)`
EXAMPLES:

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

\texttt{gen(n)}

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

**INPUT:**

- \texttt{n} - a positive integer

**OUTPUT:** MagmaElement

**EXAMPLES:**

\begin{verbatim}
sage: k.<a> = GF(9)
sage: magma(k).gen(1) # optional -- magma
a
sage: R.<s,t,w> = k[]
sage: m = magma(R) # optional -- magma
sage: m.gen(1) # optional -- magma
s
sage: m.gen(2) # optional -- magma
t
sage: m.gen(3) # optional -- magma
w
sage: m.gen(0) # optional -- magma

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

Traceback (most recent call last):
...
IndexError: list index out of range
\end{verbatim}

\texttt{gen_names()}

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

\textbf{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:**

\begin{verbatim}
sage: R.<x,zw> = QQ[]
sage: S = magma(R) # optional - magma
\end{verbatim}

(continues on next page)
sage: S.gen_names()  # optional - magma
('_sage_[...]', '_sage_[...]')
sage: magma(S.gen_names()[1])  # optional - magma
zw

gens()
Return generators for self.

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

AUTHORS:

• William Stein (2006-07-02)

EXAMPLES:

sage: magma("VectorSpace(RationalField(),3)").gens()  # optional - magma
\[
\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}
\]
sage: magma("AbelianGroup(EllipticCurve([1..5]))").gens()  # optional - magma
\[.1\]

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

OUTPUT: MagmaElement

EXAMPLES:

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

ideal(gens)
Return the ideal of self with given list of generators.

INPUT:

• gens - object or list/tuple of generators

OUTPUT:

• magma element - a Magma ideal

EXAMPLES:

sage: R = magma('PolynomialRing(RationalField())')  # optional - magma
sage: R.assign_names(['x'])  # optional - magma
sage: x = R.1  # optional - magma
sage: R.ideal([x^2 - 1, x^3 - 1])  # optional - magma
Ideal of Univariate Polynomial Ring in x over Rational Field generated by x^2 - 1, x^3 - 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
v = V.list_attributes(); v.sort() # optional - magma
print(v) # optional - magma
['Coroots', 'Involution', ..., 'p', 'ssbasis', 'weights']
```

methods (any=False)
Return signatures of all Magma intrinsics that can take self as the first argument, as strings.

INPUT:
- any – (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)
Return the quotient of self by the given object or list of generators.

INPUT:
- gens - object or list/tuple of generators

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

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

We illustrate quotienting out by an object instead of a list of generators:
```
sage: W = V.sub([ [1,2,3], [1,1,2] ]) # optional - magma
sage: V.quo(W) # optional - magma
(Full Vector space of degree 1 over Rational Field, Mapping from: Full Vector space of degree 3 over Rational Field to Full Vector space of degree 1 over Rational Field)
```

We quotient a ZZ module out by a submodule.
```
sage: V = magma.RModule(ZZ,3); V # optional - magma
RModule(IntegerRing(), 3)
sage: W, phi = V.quo([[1,2,3]]) # optional - magma
sage: W # optional - magma
```
(continues on next page)
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
10
sage: V.get_magma_attribute('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)
```
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.

21.5. Other Examples 145
This function just returns the global default Magma interface.

**EXAMPLES:**

```
sage: sage.interfaces.magma.reduce_load_Magma()
Magma
```
**INTERFACE TO THE FREE ONLINE MAGMA CALCULATOR**

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

class sage.interfaces.magma_free.MagmaFree
    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> ]
```
Chapter 22. Interface to the free online MAGMA calculator
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:

```python
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.

### 23.1 Tutorial

AUTHORS:


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

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

1. If you have a maple expression such as

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

   We can write that in sage as
Notice, there is no need to use a semicolon.

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

```sage
maple('(x^5-1)').factor()  # optional - maple
```

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

```sage
maple('(x^12-1)/(x-1)').simplify()  # optional - maple
```

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

```sage
maple('(x^12-1)').factor()  # optional - maple
```

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
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
```

instead.

If you want to load the combinatorics package for future calculations, in Sage this can be done as
sage: maple.with_package('combinat')  # optional - maple

or

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

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

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

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

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

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

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

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

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

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

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

In Sage, we write

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

More complicated programs should be put in a separate file and loaded.
**class** sage.interfaces.maple.Maple

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(.

`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:**

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

`completions(s)`

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

**EXAMPLES:**

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

`console()`

Spawn a new Maple command-line session.

**EXAMPLES:**

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

`cputime(t=None)`

Returns 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:**

```plaintext
sage: t = maple.cputime() # optional - maple
sage: t # random; optional - maple
0.02
sage: x = maple('x') # optional - maple
sage: maple.diff(x^2, x) # optional - maple
2*x
```

(continues on next page)
maple.cputime(t)  # random; optional - maple
0.0

**expect()**

Returns 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
partition10
sage: maple('bell(10)')  # optional - maple
bell10
sage: maple.with_package('combinat')  # optional - maple
sage: maple('partition(10)')  # optional - maple
[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1, 2], [1, 1, 1, 1, 1, 1, 1, 1, 1, 3], [1, 1, 1, 1, 1, 1, 1, 1, 1, 4], [1, 1, 1, 1, 1, 1, 1, 1, 1, 5], [1, 1, 1, 1, 1, 1, 1, 1, 1, 6], [1, 1, 1, 1, 1, 1, 1, 1, 1, 7], [1, 1, 1, 1, 1, 1, 1, 1, 2, 2], [1, 1, 1, 1, 1, 1, 1, 1, 2, 3], [1, 1, 1, 1, 1, 1, 1, 2, 4], [1, 1, 1, 1, 1, 1, 1, 2, 5], [1, 1, 1, 1, 1, 1, 1, 1, 2, 6], [1, 1, 1, 1, 1, 1, 1, 1, 2, 7], [1, 1, 1, 1, 1, 1, 1, 1, 2, 8], [1, 1, 1, 1, 1, 1, 1, 1, 2, 9], [1, 1, 1, 1, 1, 1, 1, 1, 2, 10]]
```
**set** (var, value)
Set the variable var to the given value.

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

**source** (s)
Display the Maple source (if possible) about s. This is the same as returning the output produced by the following Maple commands:
```
interface(verboseproc=2): print(s)
```

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

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

**with_package** (package)
Make a package of Maple procedures available in the interpreter.

**INPUT:**
- package - string

**EXAMPLES:** Some functions are unknown to Maple until you use with to include the appropriate package.
```
sage: maple.quit()         # reset maple; optional -- maple
sage: maple('partition(10)') # optional - maple
bell(10)
```

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

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 / All rights reserved. Maple is a trademark of
    \ <____ ____> Waterloo Maple Inc.
    |     Type ? for help.
    >

sage.interfaces.maple.reduce_load_Maple()
    Returns the maple object created in sage.interfaces.maple.

    EXAMPLES:

    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:

```
\begin{Verbatim}
\texttt{sage: mobj = mathematica(x^2-1)} \hspace{1cm} \# optional - mathematica
\texttt{sage: mobj.Factor()} \hspace{1cm} \# optional - mathematica
\end{Verbatim}
```

\(( -1 + x )*(1 + x)\)

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:

```
\begin{Verbatim}
\texttt{sage: mobj = mathematica(x^2-1)} \hspace{1cm} \# optional - mathematica
\texttt{sage: mobj2 = mobj.Factor(); mobj2} \hspace{1cm} \# optional - mathematica
\texttt{(-1 + x)*(1 + x)} \hspace{1cm} \texttt{Mathematica}
\texttt{sage: mobj2.parent()} \hspace{1cm} \# optional - mathematica
\texttt{Mathematica}
\texttt{sage: obj = mobj2.sage(); obj} \hspace{1cm} \# optional - mathematica
\texttt{(x + 1)*(x - 1)} \hspace{1cm} \texttt{Symbolic Ring}
\texttt{sage: obj.parent()} \hspace{1cm} \# optional - mathematica
\end{Verbatim}
```

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:

```
\begin{Verbatim}
\texttt{sage: mathematica('Factor[x^2-1]')} \hspace{1cm} \# optional - mathematica
\texttt{(-1 + x)*(1 + x)} \hspace{1cm} \texttt{Mathematica}
\texttt{sage: mathematica('Range[3]')} \hspace{1cm} \# optional - mathematica
\texttt{[1, 2, 3]} \hspace{1cm} \texttt{Mathematica}
\texttt{\_sage_repr()} \hspace{1cm} \# optional - mathematica
\end{Verbatim}
```

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 \texttt{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!)

## 24.1 Tutorial

We follow some of the tutorial from \url{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 \texttt{math} in your PATH.

### 24.1.1 Syntax

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

\begin{verbatim}
\sage: m = mathematica
\sage: a = m(1) + m(1); a
\end{verbatim}

\texttt{# optional – mathematica}

\begin{verbatim}
2
\end{verbatim}

\sage: a.parent()
\texttt{# optional – mathematica}

\texttt{Mathematica}

\begin{verbatim}
\sage: m('1+1')
\texttt{# optional – mathematica}

2
\end{verbatim}

\begin{verbatim}
\sage: m(3)**m(50)
\texttt{# optional – mathematica}

717897987691852588770249
\end{verbatim}

The following is equivalent to \texttt{Plus[2, 3]} in Mathematica:

\begin{verbatim}
\sage: m = mathematica
\sage: m(2).Plus(m(3))
\texttt{# optional – mathematica}

5
\end{verbatim}

We can also compute \texttt{7(2 + 3)}.

\begin{verbatim}
\sage: m(7).Times(m(2).Plus(m(3)))
\texttt{# optional – mathematica}

35
\end{verbatim}

\begin{verbatim}
\sage: m('7(2+3)')
\texttt{# optional – mathematica}

35
\end{verbatim}

### 24.1.2 Some typical input

We solve an equation and a system of two equations:
24.1.3 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'); eq2 = m('2x - y == 1'); 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}}
```

24.1.4 Function definitions

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

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

24.1.5 Numerical Calculations

We find the \( x \) such that \( e^x - 3x = 0 \).

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

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

```
81
```

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

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

### 24.1.6 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: g = mathematica(str(f))  # optional - mathematica
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
sage: print(g.Factor())  # optional - mathematica
3 2 3
x (x - y) (-2 x + x + y )
```

We can also factor a multivariate polynomial:

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

We factor an integer:

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

Mathematica’s ECM package is no longer available.
24.2 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
```

24.3 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
```

24.4 Complicated translations

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

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

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

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

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

```sage
m = mathematica('NewFn[x]')  # optional - mathematica
sage: m._sage_(locals={'NewFn': 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
-2.5928861754911969782 + 0.1804899720669620266 I
```

```
sage: slist = [[1, 2], 3., 4 + I]
sage: mlist = mathematica(slist); mlist  # optional - mathematica
[[1, 2], 3.0, I + 4.0]
sage: slist2 = list(mlist); slist2
# optional - mathematica
[1, 2]
sage: slist2[0].parent()  # optional - mathematica
Mathematica
sage: slist3 = mlist.sage(); slist3  # optional - mathematica
[[1, 2], 3.0, I + 4.0]
sage: mathematica('10.^80')  # optional - mathematica
1.*^80
sage: mathematica('10.^80').sage()  # optional - mathematica
1e+80
```

AUTHORS:

• William Stein (2005): first version

• Doug Cutrell (2006-03-01): Instructions for use under Cygwin/Windows.

• Felix Lawrence (2009-08-21): Added support for importing Mathematica lists and floats with exponents.

```
class sage.interfaces.mathematica.Mathematica(maxread=None, script_subdirectory=None, logfile=None, server=None, server_tmpdir=None, command=None, verbose_start=False):
Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.expect.Expect

Interface to the Mathematica interpreter.

chdir(dir)
    Change Mathematica’s current working directory.

    EXAMPLES:

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

console(readline=True)

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

get(var, ascii_art=False)
    Get the value of the variable var.

AUTHORS:

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

\texttt{help}(\texttt{cmd})

\texttt{set}(\texttt{var, value})

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

\texttt{class sage.interfaces.mathematica.MathematicaElement}(\texttt{parent, value, is_name=False, name=None})

\texttt{Bases: sage.interfaces.expect.ExpectElement}

\texttt{N}(\texttt{precision=None})

Numerical approximation by calling Mathematica’s $N[]$

Calling Mathematica’s $N[]$ function, with optional precision in decimal digits. Unlike Sage’s $n()$, $N()$ can be applied to symbolic Mathematica objects.

A workaround for trac ticket #18888 backtick issue, stripped away by get(), is included.

\textbf{Note:} The base class way up the hierarchy defines an $N$ (modeled after Mathematica’s) which overwrites the Mathematica one, and doesn’t work at all. We restore it here.

\texttt{EXAMPLES:}

\begin{verbatim}
sage: mathematica('Pi/2').N(10)  # optional -- mathematica 1.570796327
sage: mathematica('Pi').N(50)   # optional -- mathematica 3.1415926535897932384626433832795028841971693993751
sage: mathematica('Pi*x^2-1/2').N() # optional -- mathematica -0.5 + 3.14159 x
\end{verbatim}

\texttt{n}(\texttt{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 \texttt{sage.misc.functional.n()} for details.

\texttt{EXAMPLES:}

\begin{verbatim}
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
\end{verbatim}

\texttt{save_image}(\texttt{filename, ImageSize=600})

Save a mathematica graphics

\textbf{INPUT:}

• \texttt{filename} – string. The filename to save as. The extension determines the image file format.

• \texttt{ImageSize} – integer. The size of the resulting image.

\texttt{EXAMPLES:}
show (ImageSize=600)

Show a mathematica expression immediately.

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

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

show(P)  # optional - mathematica
P.show(ImageSize=800)  # optional - mathematica
Q = mathematica('Sin[x Cos[y]]/Sqrt[1-x^2]')  # optional - mathematica
Q.show()  # optional - mathematica
```

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
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
'
ans =

 4
'
sage: a = matlab(10)
# optional - matlab
sage: a**10
# optional - matlab
1.0000e+10
```

**AUTHORS:**

- William Stein (2006-10-11)

### 25.1 Tutorial

**EXAMPLES:**

```
sage: matlab('4+10')
# optional - matlab
14
sage: matlab('date')
# optional - matlab; random output
18-Oct-2006
sage: matlab('5*10 + 6')
# optional - matlab
56
sage: matlab('(6+6)/3')
# optional - matlab
4
sage: matlab('9')^2
# optional - matlab
81
sage: a = matlab(10); b = matlab(20); c = matlab(30)  
# optional - matlab
sage: avg = (a+b+c)/3 ; avg
# optional - matlab
20
sage: parent(avg)
# optional - matlab
Matlab
sage: my_scalar = matlab('3.1415')
# optional - matlab
sage: my_scalar
# optional - matlab
3.1415
```

(continues on next page)
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:

```python
sage: my_matrix = matlab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]')  # optional - matlab
sage: my_matrix(3,2)                                              # optional - matlab
4
```

Setting using parenthesis cannot work (because of how the Python language works). Use square brackets or the set function:

```python
sage: my_matrix = matlab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]')  # optional - matlab
sage: my_matrix.set(2,3, 1999)                                      # optional - matlab
sage: my_matrix                                                   # optional - matlab
[[  8  12  19]
 [  7   3 1999]
 [ 12   4  23]
 [  8   1   1]]
```

```python
class sage.interfaces.matlab.Matlab(maxread=None, script_subdirectory=None, logfile=None, server=None, server_tmpdir=None):
    Bases: sage.interfaces.expect.Expect
    Interface to the Matlab interpreter.
    EXAMPLES:
    sage: a = matlab('[ 1, 1, 2; 3, 5, 8; 13, 21, 33 ]')  # optional - matlab
    sage: b = matlab('[ 1; 3; 13]')                       # optional - matlab
    sage: c = a * b                                       # optional - matlab
    sage: print(c)                                        # optional - matlab
    [ 30
     122
     505]

    chdir(directory)
    Change MATLAB’s current working directory.
    EXAMPLES:
    sage: matlab.chdir('/')                               # optional - matlab
    sage: matlab.pwd()                                     # optional - matlab
    /

    console()  
    get(var)
    Get the value of the variable var.
    EXAMPLES:
    sage: s = matlab.eval('a = 2')  # optional - matlab
    sage: matlab.get('a')           # optional - matlab
    2

    sage2matlab_matrix_string(A)
    Return an matlab matrix from a Sage matrix.
    INPUT: A Sage matrix with entries in the rationals or reals.
    OUTPUT: A string that evaluates to an Matlab matrix.
```
EXAMPLES:

```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

```python
set(var, value)
```

Set the variable var to the given value.

```python
strip_answer(s)
```

Returns the string s with Matlab's answer prompt removed.

EXAMPLES:

```python
sage: s = 'ans =
2
'
sage: matlab.strip_answer(s)
' 2'
```

```
version()
```

```
whos()
```

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

Bases: sage.interfaces.expect.ExpectElement

```python
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.

```python
sage.interfaces.matlab.matlab_version()
```

Return the version of Matlab installed.

EXAMPLES:

```python
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:

```
sage: type(maxima)
<class 'sage.interfaces.maxima.Maxima'>
```

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

EXAMPLES: We evaluate a very simple expression in Maxima.

```
sage: maxima('3 * 5')
15
```

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

```
sage: F = maxima.factor('x^5 - y^5')
sage: F
-(y-x)*(y^4+x*y^3+x^2*y^2+x^3*y+x^4)
sage: type(F)
<class 'sage.interfaces.maxima.MaximaElement'>
```

Note that Maxima objects can also be displayed using “ASCII art”; to see a normal linear representation of any Maxima object $x$. Just use the print command: use $\text{str}(x)$.
You can always use \texttt{repr(x)} to obtain the linear representation of an object. This can be useful for moving maxima data to other systems.

\begin{Verbatim}
\texttt{sage: repr(F)}
'-(y-x)*(y^4+x*y^3+x^2*y^2+x^3*y+x^4)'
\end{Verbatim}

The \texttt{maxima.eval} command evaluates an expression in maxima and returns the result as a \texttt{string} not a maxima object.

\begin{Verbatim}
\texttt{sage: print(maxima.eval('factor(x^5 - y^5)'))}
-(y-x)*(y^4+x*y^3+x^2*y^2+x^3*y+x^4)
\end{Verbatim}

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

\begin{Verbatim}
\texttt{sage: f = maxima('x^5 - y^5')}
\texttt{sage: f^2}
(x^5-y^5)^2
\texttt{sage: f.factor()}
-(y-x)*(y^4+x*y^3+x^2*y^2+x^3*y+x^4)
\end{Verbatim}

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.

\begin{Verbatim}
\texttt{sage: maxima('sum(1/x^2, x, 1, 10)')}
1968329/1270080
\end{Verbatim}

### 26.1 Tutorial

We follow the tutorial at [http://maxima.sourceforge.net/docs/intromax/intromax.html](http://maxima.sourceforge.net/docs/intromax/intromax.html).

\begin{Verbatim}
\texttt{sage: maxima('1/100 + 1/101')}
201/10100
\end{Verbatim}

\begin{Verbatim}
\texttt{sage: a = maxima('(1 + sqrt(2))^5'); a}
(sqrt(2)+1)^5
\texttt{sage: a.expand()}
29*sqrt(2)+41
\end{Verbatim}

\begin{Verbatim}
\texttt{sage: a = maxima('(1 + sqrt(2))^5')}
\texttt{sage: float(a)}
82.01219330881975
\texttt{sage: a.numer()}
82.01219330881975
\end{Verbatim}

\begin{Verbatim}
\texttt{sage: maxima.eval('fpprec : 100')}
'100'
\end{Verbatim}
sage: a.bfloat()
8.
˓→20121933088197564152489730020812427852048438593149412212371240173412418754011041266612384955016056

sage: maxima('100!')
933262154439441526816992388562667004907159682643816214685929638521759993229915608941463976156518286

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^3*y^2+27*x*y^2+3*x^2*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}+9}{4}, b=\frac{-\sqrt{79}+1}{4}, c=\frac{\sqrt{79}+1}{10} \right], \left[ a=\frac{-\sqrt{79}+11}{4}, b=\frac{\sqrt{79}-9}{4}, c=\frac{-\sqrt{79}-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?)
k*x^3*%e^(k*x)*sin(w*x)+3*x^2*%e^(k*x)*sin(w*x)+w*x^3*%e^(k*x)*cos(w*x)

\begin{align*}
\text{sage: } f &= \int f(x) \, dx \\
\text{sage: } f &= \left(\frac{18k^3w^4 - 12k^2w^2 + 6k^4}{w^8 + 4k^2w^6 + 6k^4w^4 + 4k^6w^2 + k^8}\right) e^{kx} \sin(wx) + \left(\frac{6w^5 - 12k^2w^3 - 18k^4w}{w^8 + 4k^2w^6 + 6k^4w^4 + 4k^6w^2 + k^8}\right) e^{kx} \cos(wx)
\end{align*}

\begin{align*}
\text{sage: } f &= \frac{1}{x^2} \\
\text{sage: } f &= \int f(x, 1, 'inf') \\
\text{sage: } f &= \frac{1}{x^2} \left(\frac{1}{\sinh(kx)^4}\right) \\
\text{sage: } f &= \left(\frac{f}{x^2}\right) \left(\frac{1}{\sinh(kx)^4}\right)
\end{align*}

\begin{align*}
\text{sage: } f &= \frac{\sin(x)}{x}, 0, 10 \\
\text{sage: } f &= x + \frac{x^3}{6} + \frac{3x^5}{40} + \frac{5x^7}{112} + \frac{35x^9}{1152}
\end{align*}

\subsection*{26.2 Examples involving matrices}

We illustrate computing with the matrix whose $i, j$ entry is $i/j$, for $i, j = 1, \ldots, 4$.

\begin{align*}
\text{sage: } f &= \text{maxima.eval('f[i,j] := i/j')} \\
\text{sage: } A &= \text{maxima('genmatrix(f,4,4)')} \\
\text{sage: } A &= \text{matrix([1,1/2,1/3,1/4],[2,1,2/3,1/2],[3,3/2,1,3/4],[4,2,4/3,1])} \\
\text{sage: } A &= \text{A.determinant()} \\
\text{sage: } A &= \text{A.echelon()} \\
\text{sage: } A &= \text{A.eigenvalues()} \\
\text{sage: } A &= \text{A.eigenvectors()}
\end{align*}

We can also compute the echelon form in Sage:

\begin{align*}
\text{sage: } B &= \text{matrix(QQ, A)} \\
\text{sage: } B &= \text{B.echelon_form()} \\
\text{sage: } B &= \text{B.charpoly('x').factor()} \\
\text{sage: } B &= (x - 4) * x^3
\end{align*}

\subsection*{26.3 Laplace Transforms}

We illustrate Laplace transforms:

\begin{align*}
\text{sage: } _ &= \text{maxima.eval("f(t) := t*sin(t)"')} \\
\text{sage: } _ &= \text{maxima("laplace(f(t),t,s")')} \\
\text{sage: } _ &= \frac{(2s)/(s^2+1)^2}{945}
\end{align*}
Sage Reference Manual: Interpreter Interfaces, Release 8.7

26.4 Continued Fractions

A continued fraction \( a + \frac{1}{b + \frac{1}{c + \cdots}} \) is represented in maxima by the list \([a, b, c, \ldots]\).

\[
\text{sage: maxima("cf((1 + sqrt(5))/2)")}
\]
\[
[1,1,1,1,2]
\]

\[
\text{sage: maxima("cf ( (1 + sqrt(341))/2 )")}
\]
\[
[9,1,2,1,2,1,17,1,2,1,2,1,17,1,2,1,2,1,2,1,17,2]
\]

26.5 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 \( n \).
You can plot 3d graphs (via gnuplot):

\[
\text{sage: maxima('plot3d(x^2-y^2, [x,-2,2], [y,-2,2], [grid,12,12])')} \quad \# \text{not tested}
\]
[displays a 3 dimensional graph]

You can formally evaluate sums (note the \texttt{nusum} command):

\[
\text{sage: } S = \text{maxima('nusum(exp(1+2*i/n),i,1,n)')}
\]
\[
\text{sage: print(S)}
\]
\[
\begin{array}{c|c}
\frac{2}{n} + 3 & \frac{2}{n} + 1 \\
\hline
\frac{1}{n} & \frac{1}{n} \\
\hline
\frac{\%e - 1}{\%e + 1} & \frac{\%e - 1}{\%e + 1}
\end{array}
\]

We formally compute the limit as \( n \to \infty \) of \( 2S/n \) as follows:

\[
\text{sage: } T = S*\text{maxima('2/n')}
\]
\[
\text{sage: } T.tlimit('n','inf')
\]
\[
\%e^3-%e
\]

\section*{26.6 Miscellaneous}

Obtaining digits of \( \pi \):

\[
\text{sage: maxima.eval('fpprec : 100')}
\]
\[
'100'
\]
\[
\text{sage: maxima(pi).bfloat()}
\]
\[
3.
\rightarrow141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117067
\]

Defining functions in maxima:

\[
\text{sage: maxima.eval('fun[a] := a^2')}
\]
\[
'fun[a]:=a^2'
\]
\[
\text{sage: maxima('fun[10]')}
\]
\[
100
\]

\section*{26.7 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 \texttt{maxima.eval('integrate (exp (a*x), x, 0, inf)')}.
26.8 Latex Output

To TeX a maxima object do this:

```python
tsage: latex(maxima('sin(u) + sinh(v^2)'))
\sinh v^2+\sin u
```

Here’s another example:

```python
tsage: g = maxima('exp(3*i*x)/(6*i) + exp(i*x)/(2*i) + c')
tsage: latex(g)
-\{i, e^{3 \, i \, x}\}/6 - \{i, e^{i \, x}\}/2 + c
```

26.9 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.

```python
tsage: t = "%s"%10^10000 # ten thousand character string.
tsage: a = maxima(t)
tsage: var('Ax,Bx,By') (Ax, Bx, By) 
tsage: t = -Ax*sin(sqrt(Ax^2)/2)/(sqrt(Ax^2)*sqrt(By^2 + Bx^2))
tsage: t.limit(Ax=0, dir='+') 0
```

A long complicated input expression:

```python
tsage: maxima._eval_line('(((((((((((0) + ((1) / ((n0) ^ (0)))) + ((1) / ((n1) ^ (1)))) + ((1) / ((n2) ^ (2)))) + ((1) / ((n3) ^ (3)))) + ((1) / ((n4) ^ (4)))) + ((1) / ((n5) ^ (5)))) + ((1) / ((n6) ^ (6)))) + ((1) / ((n7) ^ (7)))) + ((1) / ((n8) ^ (8)))) + ((1) / ((n9) ^ (9))))+');
'1/n9^9+1/n8^8+1/n7^7+1/n6^6+1/n5^5+1/n4^4+1/n3^3+1/n2^2+1/n1^1'
```

Test that Maxima gracefully handles this syntax error (trac ticket #17667):

```python
tsage: maxima.eval("l == 1;") Traceback (most recent call last): ...TypeError: ...incorrect syntax: = is not a prefix operator...
```

```python
class sage.interfaces.maxima.Maxima(script_subdirectory=None, logfile=None, server=None, init_code=None)
```


Interface to the Maxima interpreter.

EXCEPTIONS:
sage: m = Maxima()
sage: m == maxima
False

clear(var)
Clear the variable named var.

EXAMPLES:

sage: maxima.set('xxxxx', '2')
sage: maxima.get('xxxxx')
'2'
sage: maxima.clear('xxxxx')
sage: maxima.get('xxxxx')
'xxxxx'

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

EXAMPLES:

sage: maxima.set('xxxxx', '2')
sage: maxima.get('xxxxx')
'2'

lisp(cmd)
Send a lisp command to Maxima.

Note: The output of this command is very raw - not pretty.

EXAMPLES:

sage: maxima.lisp("(+ 2 17)")
# random formatted output
:lisp (+ 2 17)
19
()

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

INPUT:

• var - string
• value - string

EXAMPLES:

sage: maxima.set('xxxxx', '2')
sage: maxima.get('xxxxx')
'2'

set_seed(seed=None)
http://maxima.sourceforge.net/docs/manual/maxima_10.html make_random_state (n) returns a new random state object created from an integer seed value equal to n modulo 2^32. n may be negative.

EXAMPLES:
```python
sage: m = Maxima()

sage: m.set_seed(1)

1

sage: [m.random(100) for i in range(5)]
[45, 39, 24, 68, 63]
```

```python
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:

```python
sage: maxima(3)
3
sage: maxima(cos(x)+e^234)
cos(_SAGE_VAR_x)+%e^234
```

```python
def display2d (onscreen=True)

Return the 2d string representation of this Maxima object.

EXAMPLES:

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

sage: F.display2d()
```

```python
class sage.interfaces.maxima.MaximaElementFunction (parent, name, defn, args, latex)


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:

```python
sage: maxima.function('x,y','h(x)*y')

h(x)*y
```

```python
sage.interfaces.maxima.is_MaximaElement (x)

Returns True if x is of type MaximaElement.

EXAMPLES:

```python
sage: from sage.interfaces.maxima import is_MaximaElement

sage: m = maxima(1)

sage: is_MaximaElement(m)
True

sage: is_MaximaElement(1)
False
```

```python
sage.interfaces.maxima.reduce_load_Maxima ()

Unpickle a Maxima Pexpect interface.

EXAMPLES:

```
```
sage: from sage.interfaces.maxima import reduce_load_Maxima
sage: reduce_load_Maxima()
Maxima
```

`sage.interfaces.maxima.reduce_load_Maxima_function` *(parent, defn, args, latex)*

Unpickle a Maxima function.

**EXAMPLES:**

```
sage: from sage.interfaces.maxima import reduce_load_Maxima_function
sage: f = maxima.function('x,y','sin(x+y)')
sage: _,args = f.__reduce__()
sage: g = reduce_load_Maxima_function(*args)
sage: g == f
True
```
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):

```
sage: m = Maxima()
sage: from sage.interfaces.maxima_abstract import MaximaAbstract
sage: isinstance(m,MaximaAbstract)
True
```

```
chdir (dir)
```

Change Maxima’s current working directory.

INPUT:
• dir - string
  OUTPUT: none
  EXAMPLES:

  sage: maxima.chdir('/')

completions($s$, verbose=True)
  Return all commands that complete the command starting with the string $s$. This is like typing $s$[tab] in the Maxima interpreter.

  INPUT:
  • $s$ - string
  • verbose - boolean (default: True)
  OUTPUT: array of strings
  EXAMPLES:

  sage: sorted(maxima.completions('gc', verbose=False))
  ['gcd', 'gcdex', 'gcfactor', 'gctime']

console()
  Start the interactive Maxima console. This is a completely separate maxima session from this interface. To interact with this session, you should instead use maxima.interact().

  INPUT: none
  OUTPUT: none
  EXAMPLES:

  sage: maxima.console() # not tested (since we can't)
  Maxima 5.34.1 http://maxima.sourceforge.net
  Using Lisp ECL 13.5.1
  Distributed under the GNU Public License. See the file COPYING.
  Dedicated to the memory of William Schelter.
  This is a development version of Maxima. The function bug_report()
  provides bug reporting information.
  (%i1)

  sage: maxima.interact() # this is not tested either
  --> Switching to Maxima --
  maxima: 2+2
  4
  maxima:
  --> Exiting back to Sage --

cputime($t$=None)
  Returns the amount of CPU time that this Maxima session has used.

  INPUT:
  • $t$ - float (default: None); If var{$t$} is not None, then it returns the difference between the current CPU time and var{$t$}.

  OUTPUT: float
  EXAMPLES:
```
sage: t = maxima.cputime()
sage: _ = maxima.de_solve('diff(y,x,2) + 3*x = y', ['x','y'], [1,1,1])
sage: maxima.cputime(t)  # output random
0.568913
```

**de_solve** *(de, vars, ics=None)*

Solves a 1st or 2nd order ordinary differential equation (ODE) in two variables, possibly with initial conditions.

**INPUT:**

- **de** - a string representing the ODE
- **vars** - a list of strings representing the two variables.
- **ics** - a triple of numbers [a,b1,b2] representing y(a)=b1, y'(a)=b2

**EXAMPLES:**

```
sage: maxima.de_solve('diff(y,x,2) + 3*x = y', ['x','y'], [1,1,1])
y=3*x-2*%e^(x-1)
sage: maxima.de_solve('diff(y,x,2) + 3*x = y', ['x','y'])
y=%k1*%e^x+%k2*%e^-x+3*x
sage: maxima.de_solve('diff(y,x) + 3*x = y', ['x','y'])
y=(%c-3*((-x)-1)*%e^-x)*%e^x
sage: maxima.de_solve('diff(y,x) + 3*x = y', ['x','y'], [1,1])
y=-%e^-1*(5*%e^x-3*%e*x-3*%e)
```

**de_solve_laplace** *(de, vars, ics=None)*

Solves an ordinary differential equation (ODE) using Laplace transforms.

**INPUT:**

- **de** - a string representing the ODE (e.g., de = “diff(f(x),x,2)=diff(f(x),x)+sin(x)”)  
- **vars** - a list of strings representing the variables (e.g., vars = [“x”,“f”])  
- **ics** - a list of numbers representing initial conditions, with symbols allowed which are represented by strings (eg, f(0)=1, f’(0)=2 is ics = [0,1,2])

**EXAMPLES:**

```
sage: maxima.clear('x'); maxima.clear('f')
sage: maxima.de_solve_laplace("diff(f(x),x,2) = 2*diff(f(x),x)-f(x)", ["x","f "], [0,1,2])
f(x)=x*%e^x+%e^x
```

```sage: maxima.clear('x'); maxima.clear('f')
sage: f = maxima.de_solve_laplace("diff(f(x),x,2) = 2*diff(f(x),x)-f(x)", ["x " "f"])
sage: f
f(x)=x*%e^x*('at('diff(f(x),x,1),x=0))-f(0)*x*%e^x+f(0)*%e^x
sage: print(f)
x d ! x x
f(x) = x %e (- (f(x))! ) - f(0) x %e + f(0) %e
dx !
!x = 0
```
Note: The second equation sets the values of $f(0)$ and $f'(0)$ in Maxima, so subsequent ODEs involving these variables will have these initial conditions automatically imposed.

demo $(s)$
Run Maxima’s demo for $s$.

INPUT:
• $s$ - string

OUTPUT: none

EXAMPLES:

```sage```
maxima.demo('cf') # not tested
read and interpret file: .../share/maxima/5.34.1/demo/cf.dem

At the '_' prompt, type ';' and <enter> to get next demonstration.
frac1:cf([1,2,3,4])
...```

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>, ...)
...```

dexample $(s)$
Return Maxima’s examples for $s$.

INPUT:
• $s$ - string

OUTPUT:
Maxima’s examples for $s$

EXAMPLES:

```sage```
maxima.example('arrays')
a[n]:=n*a[n-1]
a[0]:1
a[5] 120
a[n]:=n
a[6] 6```

(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: f = maxima.function('x', 'sin(x)')
sage: f(3.2)  # abs tol 2e-16
-0.058374143427579909
sage: f = maxima.function('x,y', 'sin(x)+cos(y)')
sage: f(2, 3.5)  # abs tol 2e-16
sin(2)-0.9364566872907963
sage: f
sin(x)+cos(y)
```

```
sage: g = f.integrate('z')
sage: g
(cos(y)+sin(x))*z
sage: g(1,2,3)
3*(cos(2)+sin(1))
```

The function definition can be a Maxima object:

```
sage: an_expr = maxima('sin(x)*gamma(x)')
sage: t = maxima.function('x', an_expr)
sage: t
gamma(x)*sin(x)
sage: t(2)
sin(2)
sage: float(t(2))
0.9092974268256817
sage: loads(t.dumps())
gamma(x)*sin(x)
```

help (s)
Return Maxima’s help for s.

INPUT:

• s - string

OUTPUT:

Maxima’s help for s

EXAMPLES:
sage: maxima.help('gcd')
-- Function: gcd (<p_1>, <p_2>, <x_1>, ...)

plot2d(*args)
Plot a 2d graph using Maxima / gnuplot.

maxima.plot2d(f, ['var, min, max'], options)

INPUT:

• f - a string representing a function (such as f="sin(x)") [var, xmin, xmax]

• options - an optional string representing plot2d options in gnuplot format

EXAMPLES:

sage: maxima.plot2d('sin(x)', '[x,-5,5]')  # not tested
sage: opts = '[gnuplot_term, ps], [gnuplot_out_file, "sin-plot.eps"]'
sage: maxima.plot2d('sin(x)', '[x,-5,5]', opts)  # not tested

The eps file is saved in the current directory.

plot2d_parametric(r, var, trange, nticks=50, options=None)
Plot r = [x(t), y(t)] for t = tmin...tmax using gnuplot with options.

INPUT:

• r - a string representing a function (such as r="[x(t),y(t)]")

• var - a string representing the variable (such as var = "t")

• trange - [tmin, tmax] are numbers with 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)

The eps file is saved to the current working directory.

Here is another fun plot:

sage: maxima.plot2d_parametric(['sin(5*t)', 'cos(11*t)'], 't',
                              [0,2*pi()], nticks=400)  # not tested

plot3d(*args)
Plot a 3d graph using Maxima / gnuplot.

maxima.plot3d(f, ['x, xmin, xmax'], ['y, ymin, ymax'], ['grid, nx, ny'], options)

INPUT:

• f - a string representing a function (such as f="sin(x)") [var, min, max]
• **args** should be of the form `[x, xmin, xmax]`, `[y, ymin, ymax]`, `[grid, nx, ny]`, options

**EXAMPLES:**

<table>
<thead>
<tr>
<th>Sage</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxima.plot3d</td>
<td>sage: maxima.plot3d('1 + x^3 - y^2', '[x,-2,2]', '[y,-2,2]', '[grid,12,12]') # not tested</td>
</tr>
<tr>
<td>maxima.plot3d</td>
<td>sage: maxima.plot3d('sin(x)*cos(y)', '[x,-2,2]', '[y,-2,2]', '[grid,30,30]') # not tested</td>
</tr>
<tr>
<td>opt</td>
<td>sage: opts = '[gnuplot_term, ps], [gnuplot_out_file, &quot;sin-plot.eps&quot;]'</td>
</tr>
<tr>
<td>maxima.plot3d</td>
<td>sage: maxima.plot3d('sin(x+y)', '[x,-5,5]', '[y,-1,1]', opts) # not tested</td>
</tr>
</tbody>
</table>

The eps file is saved in the current working directory.

**plot3d_parametric** *(r, vars, urange, vrange, options=None)*

Plot a 3d parametric graph with r=(x,y,z), x = x(u,v), y = y(u,v), z = z(u,v), for u = umin...umax, v = vmin...vmax using gnuplot with options.

**INPUT:**

• **x, y, z** - a string representing a function (such as x="u2+v2",...) **vars** is a list or two strings representing variables (such as vars = ["u","v"])

• **urange** - [umin, umax]

• **vrange** - [vmin, vmax] are lists of numbers with umin umax, vmin vmax

• **options** - optional string representing **plot2d** options in gnuplot format

**OUTPUT:** displays a plot on screen or saves to a file

**EXAMPLES:**

<table>
<thead>
<tr>
<th>Sage</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxima.plot3d</td>
<td>sage: maxima.plot3d_parametric([&quot;v<em>sin(u)&quot;,&quot;v</em>cos(u)&quot;,&quot;v&quot;], [&quot;u&quot;,&quot;v&quot;],[-3.2,3.2],[-2],[0,3]) # not tested</td>
</tr>
<tr>
<td>opt</td>
<td>sage: opts = '[gnuplot_term, ps], [gnuplot_out_file, &quot;sin-cos-plot.eps&quot;]'</td>
</tr>
<tr>
<td>maxima.plot3d</td>
<td>sage: maxima.plot3d_parametric([&quot;v<em>sin(u)&quot;,&quot;v</em>cos(u)&quot;,&quot;v&quot;], [&quot;u&quot;,&quot;v&quot;],[-3.2,3.2],[-2],[0,3],opts) # not tested</td>
</tr>
</tbody>
</table>

The eps file is saved in the current working directory.

Here is a torus:

<table>
<thead>
<tr>
<th>Sage</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxima.eval</td>
<td>sage: _ = maxima.eval(&quot;expr_1: cos(y)<em>(10.0+6</em>cos(x)); expr_2: sin(y)<em>(10.0+6</em>cos(x)); expr_3: 6*sin(x);&quot;)</td>
</tr>
<tr>
<td>maxima.plot3d</td>
<td>maxima.plot3d_parametric([&quot;expr_1&quot;,&quot;expr_2&quot;,&quot;expr_3&quot;], [&quot;x&quot;,&quot;y&quot;],[0,6], [0,6]) # not tested</td>
</tr>
</tbody>
</table>

Here is a Möbius strip:

<table>
<thead>
<tr>
<th>Sage</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxima.eval</td>
<td>sage: x = &quot;cos(u)<em>(3 + v</em>cos(u/2))&quot;</td>
</tr>
<tr>
<td>maxima.eval</td>
<td>sage: y = &quot;sin(u)<em>(3 + v</em>cos(u/2))&quot;</td>
</tr>
<tr>
<td>maxima.eval</td>
<td>sage: z = &quot;v*sin(u/2)&quot;</td>
</tr>
<tr>
<td>maxima.plot3d</td>
<td>maxima.plot3d_parametric([x,y,z],[&quot;u&quot;,&quot;v&quot;],[-3.1,3.2],[-1/10,1/10]) # not tested</td>
</tr>
</tbody>
</table>

**plot_list** *(ptsx, ptsy, options=None)*

Plots a curve determined by a sequence of points.

**INPUT:**

• **ptsx** - [x1,...,xn], where the xi and yi are real,

• **ptsy** - [y1,...,yn]
• **options** - a string representing maxima plot2d options.

The points are \((x_1, y_1), (x_2, y_2), \) etc.

This function requires maxima 5.9.2 or newer.

**Note:** More that 150 points can sometimes lead to the program hanging. Why?

**EXAMPLES:**

```python
sage: zeta_ptsx = [(pari(1/2 + i*I/10).zeta().real()).precision(1) for i in range (70,150)]
sage: zeta_ptsy = [(pari(1/2 + i*I/10).zeta().imag()).precision(1) for i in range (70,150)]
sage: maxima.plot_list(zeta_ptsx, zeta_ptsy)  # not tested
sage: opts='[gnuplot_preamble, "set nokey"], [gnuplot_term, ps], [gnuplot_out_file, "zeta.eps"]'
sage: maxima.plot_list(zeta_ptsx, zeta_ptsy, opts)  # not tested
```

**plot_multilist**(pts_list, options=None)

Plots a list of list of points pts_list=[pts1,pts2,…,ptsn], where each ptsi is of the form \([[x_1,y_1],[x_2,y_2],…,[x_n,y_n]]\) x’s must be integers and y’s reals options is a string representing maxima plot2d options.

**INPUT:**

- **pts_lst** - list of points; each point must be of the form \([x,y]\) where \(x\) is an integer and \(y\) is a real
- **var** - string; representing Maxima’s plot2d options

Requires maxima 5.9.2 at least.

**Note:** More that 150 points can sometimes lead to the program hanging.

**EXAMPLES:**

```python
sage: xx = [ i/10.0 for i in range (-10,10)]
sage: yy = [ i/10.0 for i in range (-10,10)]
sage: x0 = [ 0 for i in range (-10,10)]
sage: y0 = [ 0 for i in range (-10,10)]
sage: zeta_ptsx1 = [(pari(1/2+i*I/10).zeta().real()).precision(1) for i in range (10)]
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
```

**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]
```

`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 \( x^2 - 101 \)
sage: u = maxima.unit_quadratic_integer(13)
sage: u
5*a + 18
sage: u.parent()
Number Field in a with defining polynomial \( x^2 - 13 \)
```

`version()`

Return the version of Maxima that Sage includes.

INPUT: none

OUTPUT: none

EXAMPLES:

```python
sage: maxima.version()
# random
'5.41.0'
```

```python
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:

```python
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: xp = maxima(x)
sage: type(xp)
<class 'sage.interfaces.maxima.MaximaElement'>
```
```python
sage: x1 = maxima_lib(x)
sage: type(x1)
<class 'sage.interfaces.maxima_lib.MaximaLibElement'>
```

**bool()**

Convert *self* into a boolean.

**INPUT:** none

**OUTPUT:** boolean

**EXAMPLES:**

```python
sage: maxima(0).bool()
False
sage: maxima(1).bool()
True
```

**comma(args)**

Form the expression that would be written ‘*self*, args’ in Maxima.

**INPUT:**

- **args** - string

**OUTPUT:** Maxima object

**EXAMPLES:**

```python
sage: maxima('sqrt(2) + I').comma('numer')
I+1.41421356237309...
sage: maxima('sqrt(2) + I*a').comma('a=5')
5*I+sqrt(2)
```

**derivative(var='x', n=1)**

Return the n-th derivative of *self*.

**INPUT:**

- **var** - variable (default: ‘x’)
- **n** - integer (default: 1)

**OUTPUT:** n-th derivative of *self* with respect to the variable var

**EXAMPLES:**

```python
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)
```
diff(var='x', n=1)
Return the n-th derivative of self.

INPUT:
- var - variable (default: 'x')
- n - integer (default: 1)

OUTPUT: n-th derivative of self with respect to the variable var

EXAMPLES:

```
sage: f = maxima('x^2')
sage: f.diff()
2*x
sage: f.diff('x')
2*x
sage: f.diff('x', 2)
2
sage: maxima('sin(x^2)').diff('x',4)
16*x^4*sin(x^2)-12*sin(x^2)-48*x^2*cos(x^2)
```

```
sage: f = maxima('x^2 + 17*y^2')
sage: f.diff('x')
34*y*diff(y,x,1)+2*x
sage: f.diff('y')
34*y
```

dot(other)
Implements the notation self . other.

INPUT:
- other - matrix; argument to dot.

OUTPUT: Maxima matrix

EXAMPLES:

```
sage: A = maxima('matrix ([a1],[a2])')
sage: B = maxima('matrix ([b1, b2])')
sage: A.dot(B)
matrix(([a1*b1, a1*b2], [a2*b1, a2*b2]))
```

imag()
Return the imaginary part of this Maxima element.

INPUT: none

OUTPUT: Maxima real

EXAMPLES:

```
sage: maxima('2 + (2/3)*%i').imag()
2/3
```
integral (var='x', min=None, max=None)
Return the integral of self with respect to the variable x.

INPUT:
- var - variable
- min - default: None
- max - default: None

OUTPUT:
- the definite integral if xmin is not None
- an indefinite integral otherwise

EXAMPLES:

```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...
```
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
$-(\sqrt{\pi}\times%i*erf(%i))/2$
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:**

```python
sage: maxima('exp(-sqrt(x)))').nintegral('x',0,1)
(0.5284822353142306, 0.41633141378838...e-10, 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:

```
sage: maxima('a^2 + 3*a + b').subst('b=2')
a^2 + 3*a + 2
sage: maxima('a^2 + 3*a + b').subst('a=17')
b + 340
sage: maxima('a^2 + 3*a + b').subst('a=17, b=2')
342
```

```
class sage.interfaces.maxima_abstract.MaximaAbstractElementFunction(parent, name, defn, args, latex)

Bases: sage.interfaces.maxima_abstract.MaximaAbstractElement

Create a Maxima function with the parent parent, name name, definition defn, arguments args and latex representation latex.

INPUT:

- parent - an instance of a concrete Maxima interface
- name - string
- defn - string
- args - string; comma separated names of arguments
- latex - string

OUTPUT: Maxima function

EXAMPLES:

```
sage: maxima.function('x,y','e^cos(x)')
e^cos(x)
sage: f.arguments() ['x', 'y']
sage: f.arguments(split=False)
['x', 'y']
```

```
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)
```

(continues on next page)
definition()  
Returns the definition of this Maxima function as a string. 

INPUT: none  
OUTPUT: string  

EXAMPLES:  

```
sage: f = maxima.function('x,y','sin(x+y)')
sage: f.definition()
'sin(x+y)'
```  

integral(var)  
Returns the integral of self with respect to the variable var.  

INPUT:  
- var - a variable  

OUTPUT: Maxima function  

Note that integrate is an alias of integral.  

EXAMPLES:  

```
sage: x,y = var('x,y')
sage: f = maxima.function('x','sin(x)')
sage: f.integral(x)
-cos(x)
sage: f.integral(y)
sin(x)*y
```  

integrate(var)  
Returns the integral of self with respect to the variable var.  

INPUT:  
- var - a variable  

OUTPUT: Maxima function  

Note that integrate is an alias of integral.  

EXAMPLES:  

```
sage: x,y = var('x,y')
sage: f = maxima.function('x','sin(x)')
sage: f.integrate(x)
-cos(x)
sage: f.integrate(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:

```python
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:

```python
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
```

**class** sage.interfaces.maxima_lib.MaximaLib

**Bases:** sage.interfaces.maxima_abstract.MaximaAbstract

Interface to Maxima as a Library.

**INPUT:** none

**OUTPUT:** Maxima interface as a Library

**EXAMPLES:**

```
sage: from sage.interfaces.maxima_lib import MaximaLib, maxima_lib
sage: isinstance(maxima_lib,MaximaLib)
True
```

Only one such interface can be instantiated:

```
sage: Maximalib()
Traceback (most recent call last):
  ...
RuntimeError: Maxima interface in library mode can only be instantiated once
```

**clear**(var)

Clear the variable named var.

**INPUT:**

- var - string

**OUTPUT:** none

**EXAMPLES:**

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib.set('xxxxx', '2')
sage: maxima_lib.get('xxxxx')
'2'
sage: maxima_lib.clear('xxxxx')
sage: maxima_lib.get('xxxxx')
'xxxxx'
```

**eval**(line, locals=None, reformat=True, **kwds)

Evaluate the line in Maxima.

**INPUT:**
• **line** - string; text to evaluate
• **locals** - None (ignored); this is used for compatibility with the Sage notebook’s generic system interface.
• **reformat** - boolean; whether to strip output or not
• ****kwds** - All other arguments are currently ignored.

**OUTPUT:** string representing Maxima output

**EXAMPLES:**

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib._eval_line('1+1')
'2'
sage: maxima_lib._eval_line('1+1;')
'2'
sage: maxima_lib._eval_line('1+1$')
''
sage: maxima_lib._eval_line('randvar : cos(x)+sin(y)$')
''
sage: maxima_lib._eval_line('randvar')
'sin(y)+cos(x)'
```

**get**(var)

Get the string value of the variable var.

**INPUT:**

• var - string

**OUTPUT:** string

**EXAMPLES:**

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib.set('xxxxx', '2')
sage: maxima_lib.get('xxxxx')
'2'
```

**lisp**(cmd)

Send a lisp command to maxima.

**INPUT:**

• cmd - string

**OUTPUT:** ECL object

**Note:** The output of this command is very raw - not pretty.

**EXAMPLES:**

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib.lisp("(+ 2 17)")
<ECL: 19>
```

**set**(var, value)

Set the variable var to the given value.

**INPUT:**
• var - string
• value - string

OUTPUT: none

EXAMPLES:

```python
sage: from sage.interfaces.maxima_lib import maxima_lib
gsage: maxima_lib.set('xxxxx', '2')
gsage: maxima_lib.get('xxxxx')
'2'
```

`sr_integral(*args)`
Helper function to wrap calculus use of Maxima's integration.

```python
sage: integrate(sgn(x) - sgn(1-x), x)
abs(x - 1) + abs(x)
```

This is a known bug in Sage symbolic limits code, see trac ticket #17892 and https://sourceforge.net/p/maxima/bugs/3237/

```python
sage: integrate(1 / (1 + abs(x-5)), x, -5, 6)  # not tested -- known bug
log(11) + log(2)
```

```python
sage: integrate(1/(1 + abs(x)), x)
1/2*(log(x + 1) + log(-x + 1))*sgn(x) + 1/2*log(x + 1) - 1/2*log(-x + 1)
```

```python
sage: integrate(cos(x + abs(x)), x)
-1/2*x*sgn(x) + 1/4*(sgn(x) + 1)*sin(2*x) + 1/2*x
```

The last example relies on the following simplification:

```python
sage: maxima("realpart(signum(x))")
signum(x)
```

An example from sage-support thread e641001f8b8d1129:

```python
sage: f = e^(-x^2/2)/sqrt(2*pi) * sgn(x-1)
sage: integrate(f, x, -Infinity, Infinity)
-erf(1/2*sqrt(2))
```

From trac ticket #8624:

```python
sage: integral(abs(cos(x))*sin(x),(x,pi/2,pi))
1/2
```

```python
sage: integrate(sqrt(x + sqrt(x)), x).canonicalize_radical()
1/12*((8*x - 3)*x^(1/4) + 2*x^(3/4))*sqrt(sqrt(x) + 1) + 1/8*log(sqrt(sqrt(x) + 1) + x^(1/4)) - 1/8*log(sqrt(sqrt(x) + 1) - x^(1/4))
```

And trac ticket #11594:

```python
sage: integrate(abs(x^2 - 1), x, -2, 2)
4
```

This definite integral returned zero (incorrectly) in at least Maxima 5.23. The correct answer is now given (trac ticket #11591):
Sometimes one needs different simplification settings, such as `radexpand`, to compute an integral (see trac ticket #10955):

```python
sage: f = sqrt(x + 1/x^2)
sage: maxima = sage.calculus.calculus.maxima
sage: maxima('radexpand')
true
sage: integrate(f, x)
```

The following integral was computed incorrectly in versions of Maxima before 5.27 (see trac ticket #12947):

```python
sage: a = integrate(x*cos(x^3),(x,0,1/2)).n()
sage: a.real()
0.124756040961038
sage: a.imag().abs() < 3e-17
True
```

**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.

```python
sage: x, y, k, n = var('x, y, k, n')
sage: sum(binomial(n,k) * x^k * y^(n-k), k, 0, n)
(x + y)^n
sage: q, a = var('q, a')
sage: sum(a*q^k, k, 0, oo)
Traceback (most recent call last):
  ...
ValueError: Computation failed since Maxima requested additional constraints; using the 'assume' command before evaluation *may* help
(example of legal syntax is 'assume(abs(q)-1>0)', see `assume?`
for more details)
Is abs(q)-1 positive, negative or zero?
sage: assume(q > 1)
sage: sum(a*q^k, k, 0, oo)
Traceback (most recent call last):
  ...
ValueError: Sum is divergent.
sage: forget()
```
(continues on next page)
Taking the sum of all natural numbers informs us that the sum is divergent. Maxima (before 5.29.1) used to ask questions about \( m \), leading to a different error (see trac ticket #11990):

```
sage: m = var('m')
sage: sum(m, m, 0, infinity)
Traceback (most recent call last):
  ...  
ValueError: Sum is divergent.
```

An error with an infinite sum in Maxima (before 5.30.0, see trac ticket #13712):

```
sage: n = var('n')
sage: sum(1/((2*n-1)^2*(2*n+1)^2*(2*n+3)^2), n, 0, oo)
3/256*pi^2
```

Maxima correctly detects division by zero in a symbolic sum (see trac ticket #11894):

```
sage: sum(1/(m^4 + 2*m^3 + 3*m^2 + 2*m)^2, m, 0, infinity)
Traceback (most recent call last):
  ...  
RuntimeError: ECL says: Error executing code in Maxima: Zero to negative_\rightarrow power computed.
```

Similar situation for trac ticket #12410:

```
sage: x = var('x')
sage: sum(1/x*(-1)^x, x, 0, oo)
Traceback (most recent call last):
  ...  
RuntimeError: ECL says: Error executing code in Maxima: Zero to negative_\rightarrow power computed.
```

**sr_tlimit** (expr, v, a, dir=None)

Helper function to wrap calculus use of Maxima’s Taylor series limits.

**class** sage.interfaces.maxima_lib.MaximaLibElement (parent, value, is_name=False, name=None)

Element of Maxima through library interface.

**EXAMPLES:**

Elements of this class should not be created directly. The targeted parent should be used instead:

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib(4)
4
sage: maxima_lib(log(x))
log(_SAGE_VAR_x)
```
display2d(onscreen=True)

Return the 2d representation of this Maxima object.

INPUT:

• onscreen - boolean (default: True); whether to print or return

OUTPUT:

The representation is printed if onscreen is set to True and returned as a string otherwise.

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: F = maxima_lib('x^5 - y^5').factor()
sage: F.display2d()
4 3 2 2 3 4
   -(y - x) (y + x y + x y + x y + x )
```

ecl()

Return the underlying ECL object of this MaximaLib object.

INPUT: none

OUTPUT: ECL object

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib(x+cos(19)).ecl()
<ECL: ((MPLUS SIMP) ((%COS SIMP) 19) |$_SAGE_VAR_x|)>
```

to_poly_solve(vars, options="")

Use Maxima’s to_poly_solver package.

INPUT:

• vars - symbolic expressions
• options - string (default="")

OUTPUT: Maxima object

EXAMPLES:

The zXXX below are names for arbitrary integers and subject to change:

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: sol = maxima_lib(sin(x) == 0).to_poly_solve(x)
sage: sol.sage()
[[x == pi*z...]]
```

class sage.interfaces.maxima_lib.MaximaLibElementFunction(parent, name, defn, args, latex)


sage.interfaces.maxima_lib.dummy_integrate(expr)

We would like to simply tie Maxima’s integrate to sage.calculus.calculus.dummy_integrate, but we’re being imported there so to avoid circularity we define it here.

INPUT:

• expr - ECL object; a Maxima %INTEGRATE expression
OUTPUT: symbolic expression

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib, dummy_integrate
sage: f = maxima_lib('f(x)').integrate('x')
sage: f.ecl()
<ECL: ((%INTEGRATE SIMP) (($F SIMP) $X) $X)>
sage: dummy_integrate(f.ecl())
integrate(f(x), x)
```

```
sage: f = maxima_lib('f(x)').integrate('x',0,10)
sage: f.ecl()
<ECL: ((%INTEGRATE SIMP) (($F SIMP) $X) $X 0 10)>
sage: dummy_integrate(f.ecl())
integrate(f(x), x, 0, 10)
```

```
sage.interfaces.maxima_lib.is_MaximaLibElement(x)
Returns True if x is of type MaximaLibElement.

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib, is_MaximaLibElement
sage: m = maxima_lib(1)
sage: is_MaximaLibElement(m)
True
sage: is_MaximaLibElement(1)
False
```

```
sage.interfaces.maxima_lib.max_at_to_sage(expr)
Special conversion rule for AT expressions.

INPUT:

• expr - ECL object; a Maxima AT expression

OUTPUT: symbolic expression

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib, max_at_to_sage
sage: a=maxima_lib("'at(f(x,y,z),[x=1,y=2,z=3])")
sage: a
'at(f(x,y,z),[x=1,y=2,z=3])
sage: max_at_to_sage(a.ecl())
f(1, 2, 3)
sage: a=maxima_lib("'at(f(x,y,z),x=1)"")
sage: a
'at(f(x,y,z),x=1)
sage: max_at_to_sage(a.ecl())
f(1, y, z)
```

```
sage.interfaces.maxima_lib.max_harmonic_to_sage(expr)
EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib, max_to_sr
sage: c=maxima_lib(harmonic_number(x,2))
sage: c.ecl()<ECL: (($GEN_HARMONIC_NUMBER SIMP) 2 |$_SAGE_VAR_x|)>
```
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
codeblock
sage: f = maxima_lib('f(x)')
sage: f.ecl()  
<ECL: (($F SIMP) $X)>
sage: max_to_sr(f.ecl())
f(x)
```

sage.interfaces.maxima_lib.max_to_string(s)
Return the Maxima string corresponding to this ECL object.

INPUT:
  • s - ECL object

OUTPUT: string

EXAMPLES:

```python
sage: from sage.interfaces.maxima_lib import maxima_lib, max_to_string
codeblock
sage: ecl = maxima_lib(cos(x)).ecl()
sage: max_to_string(ecl)
'cos(_SAGE_VAR_x)'
```

sage.interfaces.maxima_lib.mdiff_to_sage(expr)
Special conversion rule for %DERIVATIVE expressions.

INPUT:
  • expr - ECL object; a Maxima %DERIVATIVE expression

OUTPUT: symbolic expression

EXAMPLES:

```python
sage: from sage.interfaces.maxima_lib import maxima_lib, mdiff_to_sage
codeblock
sage: f = maxima_lib('f(x)').diff('x',4)
sage: f.ecl()  
<ECL: (%DERIVATIVE SIMP) ((S$ SIMP) $X) $X 4)>
sage: mdiff_to_sage(f.ecl())
diff(f(x), x, x, x, x)
```

sage.interfaces.maxima_lib.mlist_to_sage(expr)
Special conversion rule for MLIST expressions.

INPUT:
  • expr - ECL object; a Maxima MLIST expression (i.e., a list)
OUTPUT: a Python list of converted expressions.

EXAMPLES:

```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_z^8+_SAGE_VAR_y^3+_SAGE_VAR_x*щен
→_SAGE_VAR_y)/(_SAGE_VAR_x^6*_SAGE_VAR_y)
sage: c.ecl()
<ECL: ((MRAT SIMP (|$_SAGE_VAR_x| |$_SAGE_VAR_y| |$_SAGE_VAR_z|)
sage: mrat_to_sage(c.ecl())
(x^6*z^8 + y*z^9 + y^3 + x*y)/(x^6*y)
```

`sage.interfaces.maxima_lib.parse_max_string(s)`  
Evaluate string in Maxima without any further simplification.

**INPUT:**
• **s** - string

**OUTPUT:** ECL object

**EXAMPLES:**

```python
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:**

```python
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:**

```python
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:**

```python
sage: from sage.interfaces.maxima_lib import sage_rat
sage: sage_rat(1, 7)
1/7
```
sage.interfaces.maxima_lib.sr_to_max(expr)
Convert a symbolic expression into a Maxima object.

INPUT:
- expr - symbolic expression

OUTPUT: ECL object

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import sr_to_max
sage: var('x')
x
sage: sr_to_max(x)
<ECL: $X>
sage: sr_to_max(cos(x))
<ECL: ((%COS) $X)>
sage: f = function('f')(x)
sage: sr_to_max(f.diff())
<ECL: ((%DERIVATIVE) (($F) $X) $X 1)>
```

sage.interfaces.maxima_lib.stdout_to_string(s)
Evaluate command s and catch Maxima stdout (not the result of the command!) into a string.

INPUT:
- s - string; command to evaluate

OUTPUT: string

This is currently used to implement `display2d()`.

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import stdout_to_string
sage: stdout_to_string('1+1')
''
sage: stdout_to_string('disp(1+1)')
'2\n\n'
```
INTERFACE TO MUPAD

AUTHOR:

- Mike Hansen
- William Stein

You must have the optional commercial MuPAD interpreter installed and available as the command code{mupkern} in your PATH in order to use this interface. You do not have to install any optional sage packages.

```python
class sage.interfaces.mupad.Mupad(maxread=None, script_subdirectory=None, server=None, server_tmpdir=None, logfile=None):
    Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.expect.Expect

    Interface to the MuPAD interpreter.

    completions(string, strip=False)
    
    EXAMPLES:
    
    sage: mupad.completions('linal')  # optional - mupad
    ['linalg']

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

    EXAMPLES:
    
    sage: mupad.console()  #not tested

    *----* MuPAD Pro 4.0.2 -- The Open Computer Algebra System
    |/ |
    *----* | Copyright (c) 1997 - 2007 by SciFace Software
    | **--** All rights reserved.
    | /  |
    *----* Licensed to: ...

cputime(t=None)
    
    EXAMPLES:
    
    sage: t = mupad.cputime()  #random, optional - MuPAD
    0.11600000000000001

eval(code, strip=True, **kwds)
    
    EXAMPLES:
```
**sage**: mupad.eval('2+2')  # optional - mupad
\[
4
\]

**expect()**

EXAMPLES:

```
sage: a = mupad(1)  # optional - mupad
sage: mupad.expect()  # optional - mupad
<pexpect.spawn instance at 0x...>
```

**get(var)**

Get the value of the variable var.

EXAMPLES:

```
sage: mupad.set('a', 4)  # optional - mupad
sage: mupad.get('a').strip()  # optional - mupad
'4'
```

**set(var, value)**

Set the variable var to the given value.

EXAMPLES:

```
sage: mupad.set('a', 4)  # optional - mupad
sage: mupad.get('a').strip()  # optional - mupad
'4'
```

```python
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:
```python
sage: from sage.interfaces.mupad import reduce_load_mupad
sage: reduce_load_mupad()
Mupad
```
sage.interfaces.mwrank.Mwrank(\texttt{options=``
, server=None, server_tmpdir=None})

Create and return an mwrank interpreter, with given options.

\begin{itemize}
\item \texttt{options} - string; passed when starting mwrank. The format is:
\end{itemize}

\begin{itemize}
\item \texttt{-h} \hspace{.5em} \text{help} \hspace{1.5em} \text{prints this info and quits}
\item \texttt{-q} \hspace{.5em} \text{quiet} \hspace{1.5em} \text{turns OFF banner display and prompt}
\item \texttt{-v n} \hspace{.5em} \text{verbosity} \hspace{1.5em} \text{sets verbosity to n (default=1)}
\item \texttt{-o} \hspace{.5em} \text{PARI/GP output} \hspace{1.5em} \text{turns ON extra PARI/GP short output (default is OFF)}
\item \texttt{-p n} \hspace{.5em} \text{precision} \hspace{1.5em} \text{sets precision to n decimals (default=15)}
\item \texttt{-b n} \hspace{.5em} \text{quartic bound} \hspace{1.5em} \text{bound on quartic point search (default=10)}
\item \texttt{-x n} \hspace{.5em} \text{n aux} \hspace{1.5em} \text{number of aux primes used for sieving (default=6)}
\item \texttt{-l} \hspace{.5em} \text{list} \hspace{1.5em} \text{turns ON listing of points (default ON unless v=0)}
\item \texttt{-s} \hspace{.5em} \text{selmer_only} \hspace{1.5em} \text{if set, computes Selmer rank only (default: not set)}
\item \texttt{-d} \hspace{.5em} \text{skip_2nd_descent} \hspace{1.5em} \text{if set, skips the second descent for curves}
\item \texttt{-S n} \hspace{.5em} \text{sat_bd} \hspace{1.5em} \text{upper bound on saturation primes (default=100, -1 for automatic)}
\end{itemize}

\textbf{EXAMPLES:}

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

\textbf{class} \hspace{.5em} \texttt{sage.interfaces.mwrank.Mwrank Class(\texttt{options=``
, server=None, server_tmpdir=None})}

\textbf{Bases:} \texttt{sage.interfaces.expect.Expect}

Interface to the Mwrank interpreter.

\texttt{console()}

Start the mwrank console.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: mwrank.console() # not tested: expects console input
Program mwrank: ...
\end{verbatim}

\texttt{eval(s,**kwds)}

Return mwrank’s output for the given input.

\textbf{INPUT:}
• \(s\) (str) - a Sage object which when converted to a string gives valid input to \texttt{mwrank}. The conversion is done by \texttt{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.

\textbf{Note:} If a RuntimeError exception is raised, then the mwrank interface is restarted and the command is retried once.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: mwrank.eval('12 3 4 5 6') 'Curve [12,3,4,5,6] :...'
sage: mwrank.eval('12,3,4,5,6')  'Curve [12,3,4,5,6] :...'
sage: mwrank.eval([12, 3, 4, 5, 6])  'Curve [12,3,4,5,6] :...'
sage: mwrank.eval((12, 3, 4, 5, 6))  'Curve [12,3,4,5,6] :...'
\end{verbatim}

sage.interfaces.mwrank.mwrank_console()

Start the mwrank console.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: mwrank_console()  # not tested: expects console input
Program mwrank: ...
\end{verbatim}

sage.interfaces.mwrank.validate_mwrank_input(s)

Returns a string suitable for mwrank input, or raises an error.

\textbf{INPUT:}

• \(s\) – one of the following:
  - a list or tuple of 5 integers \([a1,a2,a3,a4,a6]\) or \((a1,a2,a3,a4,a6)\)
  - a string of the form ‘\([a1,a2,a3,a4,a6]\)’ or ‘a1 a2 a3 a4 a6’ where a1, a2, a3, a4, a6 are integers

\textbf{OUTPUT:}

For valid input, a string of the form ‘\([a1,a2,a3,a4,a6]\)’. For invalid input a ValueError is raised.

\textbf{EXAMPLES:}

A list or tuple of 5 integers:

\begin{verbatim}
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)
\end{verbatim}
sage: validate_mwrank_input([1,2,3,4,i])
Traceback (most recent call last):
...
ValueError: [1, 2, 3, 4, I] is not valid input to mwrank (entries should be integers)

A string of the form ‘[a1,a2,a3,a4,a6]’ with any whitespace and integers ai:

sage: validate_mwrank_input('0 -1 1 -7 6')
'\[0,-1,1,-7,6\]'

sage: validate_mwrank_input('0
-1
1
-7
6')
'\[0,-1,1,-7,6\]'

sage: validate_mwrank_input('0	-1	1	0	0')
'\[0,-1,1,0,0\]'

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
```

31.1 Computation of Special Functions

Octave implements computation of the following special functions (see the maxima and gp interfaces for even more special functions):

- airy
  Airy functions of the first and second kind, and their derivatives.
  airy(0,x) = Ai(x), airy(1,x) = Ai'(x), airy(2,x) = Bi(x), airy(3,x) = Bi'(x)

- besselj
  Bessel functions of the first kind.

- bessely
  Bessel functions of the second kind.

- besseli
  Modified Bessel functions of the first kind.

- besselk
  Modified Bessel functions of the second kind.

- besselh
  Compute Hankel functions of the first \( k = 1 \) or second \( k = 2 \) kind.

- beta
  The Beta function,
  \[ \beta(a, b) = \frac{\Gamma(a) \times \Gamma(b)}{\Gamma(a + b)}. \]

- betainc
  The incomplete Beta function,
The error function, \texttt{erfinv}  
The inverse of the error function.
\texttt{gamma}  
The Gamma function, \texttt{gammainc}  
The incomplete gamma function,

For example,

\begin{verbatim}
\texttt{sage: octave("airy(3,2)")} # optional - octave
4.10068
\texttt{sage: octave("beta(2,2)")} # optional - octave
0.166667
\texttt{sage: octave("betainc(0.2,2,2)")} # optional - octave
0.104
\texttt{sage: octave("besselh(0,2)")} # optional - octave
(0.223891,0.510376)
\texttt{sage: octave("besselh(0,1)")} # optional - octave
(0.765198,0.088257)
\texttt{sage: octave("besseli(1,2)")} # optional - octave
1.59064
\texttt{sage: octave("besselk(1,2)")} # optional - octave
0.576725
\texttt{sage: octave("besselk(1,2)")} # optional - octave
0.139866
\texttt{sage: octave("erf(0)")} # optional - octave
0
\texttt{sage: octave("erf(0)")} # optional - octave
0.842701
\texttt{sage: octave("erfinv(0.842)")} # optional - octave
0.998315
\texttt{sage: octave("gamma(1.5)")} # optional - octave
0.886227
\texttt{sage: octave("gammainc(1.5,1)")} # optional - octave
0.77687
\end{verbatim}

31.2 Tutorial

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage: octave('4+10')} # optional - octave
14
\texttt{sage: octave('date')} # optional - octave; random output
18-Oct-2007
\texttt{sage: octave('5*10 + 6')} # optional - octave
56
\texttt{sage: octave('(6+6)/3')} # optional - octave
4
\texttt{sage: octave('9')^2} # optional - octave
81
\texttt{sage: a = octave(10); b = octave(20); c = octave(30)} # optional - octave
\texttt{sage: avg = (a+b+c)/3} # optional - octave
\texttt{sage: avg} # optional - octave
\end{verbatim}
20
sage: parent(avg)  # optional - octave

Octave

sage: my_scalar = octave('3.1415')  # optional - octave
sage: my_scalar
3.1415
sage: my_vector1 = octave('[1,5,7]')  # optional - octave
sage: my_vector1
1 5 7
sage: my_vector2 = octave('[1;5;7]')  # optional - octave
sage: my_vector2
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*c = b # optional - octave
'c =

 1
 7.21645e-16
 -7.21645e-16

'
sage: octave.eval("c")  # optional - octave
'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"

call

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).
Pressing ctrl-d exits the octave console and returns you to Sage. octave, like Sage, remembers its history from one session to another.

**de_system_plot** *(f, ics, trange)*
Plot (using octave’s interface to gnuplot) the solution to a 2 × 2 system of differential equations.

**INPUT:**
- *f* - a pair of strings representing the differential equations; The independent variable must be called x and the dependent variable must be called y.
- *ics* - a pair [x0,y0] such that x(t0) = x0, y(t0) = y0
- *trange* - a pair [t0,t1]

**OUTPUT:** a gnuplot window appears

**EXAMPLES:**
```
sage: octave.de_system_plot(["x+y","x-y"], [1,-1], [0,2]) # not tested -- does this actually work (on OS X it fails for me -- William Stein, 2007-10)
```
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:**
```
sage: octave.set('x', '2') # optional - octave
sage: octave.get('x') # optional - octave
'2'
```

**quit**(verbose=False)

**EXAMPLES:**
```
sage: o = Octave()
sage: o._start() # optional - octave
sage: o.quit(True) # optional - octave
Exiting spawned Octave process.
```

**sage2octave_matrix_string**(A)
Return an octave matrix from a Sage matrix.

**INPUT:** A Sage matrix with entries in the rationals or reals.

**OUTPUT:** A string that evaluates to an Octave matrix.

**EXAMPLES:**
sage: M33 = MatrixSpace(QQ,3,3)
sage: A = M33([[1,2,3,4,5,6,7,8,0]])
sage: octave.sage2octave_matrix_string(A)  # optional - octave
'[1, 2, 3; 4, 5, 6; 7, 8, 0]'

AUTHORS:

- David Joyner and William Stein

set\( (\texttt{var}, \texttt{value}) \)
Set the variable \texttt{var} to the given \texttt{value}.

EXAMPLES:

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

set_seed\( (\texttt{seed=\texttt{None}}) \)
Set the seed for the random number generator for this octave interpreter.

EXAMPLES:

```
sage: o = Octave()  # optional - octave
sage: o.set_seed(1)  # optional - octave
1
sage: [o.rand() for i in range(5)]  # optional - octave
[ 0.134364, 0.847434, 0.763775, 0.255069, 0.495435]
```

solve_linear_system\( (A, b) \)
Use octave to compute a solution \(x\) to \(A\times x = b\), as a list.

INPUT:

- \(A\) – \(m\times n\) 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 \(M\times x = b\)

EXAMPLES:

```
sage: M33 = MatrixSpace(QQ,3,3)
sage: A = M33([[1,2,3,4,5,6,7,8,0]])
sage: V3 = VectorSpace(QQ,3)
sage: b = V3([1,2,3])
sage: octave.solve_linear_system(A,b)  # optional - octave
(-0.33333299999999999, 0.66666700000000001, -3.5236600000000002e-18)
```

AUTHORS:

- David Joyner and William Stein

version\( () \)
Return the version of Octave.

OUTPUT: string

EXAMPLES:
sage: v = octave.version()  # optional - octave
sage: v                # optional - octave; random
'2.13.7'

sage: import re
sage: assert re.match(r"\d+\.\d+\.\d+", v)  is not None # optional - octave

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> [ct1-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
```
CHAPTER

THIRTYTWO

INTERFACE TO PHC.

PHC computes numerical information about systems of polynomials over the complex numbers.

PHC implements polynomial homotopy methods to exploit structure in order to better approximate all isolated solutions. The package also includes extra tools to handle positive dimensional solution components.

AUTHORS:

- PHC was written by J. Verschelde, R. Cools, and many others (?)
- William Stein and Kelly ?? – first version of interface to PHC
- Marshall Hampton – second version of interface to PHC
- Marshall Hampton and Alex Jokela – third version, path tracking

class sage.interfaces.phc.PHC

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: v.solutions() # optional -- phc
sage: v.solutions.sort() # optional -- phc
sage: v.solutions # optional -- phc
[[[-1.00000000000000*I, 1.00000000000000*I], [1.00000000000000*I, -1.00000000000000*I]]
sage: sol_dict = v.solutions_dicts() # optional -- phc
sage: x_sols_from_dict = [d[x] for d in sol_dict] # optional -- phc
sage: x_sols_from_dict.sort(); x_sols_from_dict # optional -- phc
[-1.00000000000000*I, 1.00000000000000*I]
sage: residuals = [[test_equation.change_ring(CDF).subs(sol) for test_equation in testsys] for sol in v.solutions_dicts()] # optional -- phc
sage: residuals # optional -- phc
[[0, 0], [0, 0]]
```

blackbox(polys, input_ring, verbose=False)

Returns 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)  # optional -- phc
sage: len(sol.solutions())
30
```

mixed_volume(polys, verbose=False)

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]
sage: phc.mixed_volume(test_sys)  # optional -- phc
4
```

path_track(start_sys, end_sys, input_ring, c_skew=0.001, saved_start=None)

This function computes homotopy paths between the solutions of start_sys and end_sys.

INPUT:
• start_sys – a square polynomial system, given as a list of polynomials
• end_sys – same type as start_sys
• input_ring – for coercion of the variables into the desired ring.
• c_skew – optional. the imaginary part of homotopy multiplier; nonzero values are often necessary to avoid intermediate path collisions
• saved_start – optional. A phc output file. If not given, start system solutions are computed via the phc.blackbox function.

OUTPUT:
• a list of paths as dictionaries, with the keys variables and t-values on the path.

EXAMPLES:

```python
sage: from sage.interfaces.phc import *
sage: R2.<x,y> = PolynomialRing(QQ,2)
sage: start_sys = [x^6-y^2,y^5-1]
(continues on next page)```
sage: sol = phc.blackbox(start_sys, R2)      # optional -- phc
sage: start_save = sol.save_as_start()      # optional -- phc
sage: end_sys = [x^7-2, y^5-x^2]             # optional -- phc
sage: sol_paths = phc.path_track(start_sys, end_sys, R2, saved_start = start_save) # optional -- phc
sage: len(sol_paths)                         # optional -- phc
30

plot_paths_2d(start_sys, end_sys, input_ring, c_skew=0.001, endpoints=True, saved_start=None, rand_colors=False)

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:

sage: from sage.interfaces.phc import *
sage: from sage.structure.sage_object import SageObject
sage: R2.<x,y> = PolynomialRing(QQ,2)
sage: start_sys = [x^5-y^2,y^5-1]
sage: sol = phc.blackbox(start_sys, R2)      # optional -- phc
sage: start_save = sol.save_as_start()      # optional -- phc
sage: end_sys = [x^5-25,y^5-x^2]             # optional -- phc
sage: testing = phc.plot_paths_2d(start_sys, end_sys, R2) # optional -- phc
sage: type(testing)                         # optional -- phc (normally use plot here)
<class 'sage.plot.graphics.Graphics'>

start_from(start_filename_or_string, polys, input_ring, path_track_file=None, verbose=False)

This computes solutions starting from a phcspack solution file.

INPUT:

- start_filename_or_string – the filename for a phcspack 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]
sage: sol = phc.blackbox(start_sys, R2)  # optional -- phc
sage: start_save = sol.save_as_start()  # optional -- phc
sage: end_sys = [x^7-2,y^5-x^2]  # optional -- phc
sage: sol = phc.start_from(start_save, end_sys, R2)  # optional -- phc
sage: len(sol.solutions())  # optional -- phc
30
```

### class sage.interfaces.phc.PHC_Object(output_file_contents, input_ring)

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]
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'
```

### classified_solution_dicts()

Returns 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:
solution_dicts (get_failures=False)

Returns 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:

```python
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)

Returns 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:

```python
sage: from sage.interfaces.phc import *
sage: R2.<x1,x2> = PolynomialRing(QQ,2)
sage: test_sys = [x1^5-x1*x2^2-1, x2^5-x1*x2-1]
sage: sol = phc.blackbox(test_sys, R2)  # optional -- phc
sage: len(sol.solutions())  # optional -- phc
25
```
variable_list()

Returns 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)

Returns 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)

Returns a list of dictionaries of variable:value (key:value) pairs. Only used internally; see the solution_dict function in the PHC_Object class definition for details.

INPUT:
- output_file_contents – phc solution output as a string
- input_ring – a PolynomialRing that variable names can be coerced into

OUTPUT:
- a list of dictionaries of solutions

EXAMPLES:

```
sage: from sage.interfaces.phc import *
sage: R2.<x1,x2> = PolynomialRing(QQ,2)
sage: test_sys = [(x1-1)^5-x2, (x2-1)^5-1]
sage: sol = phc.blackbox(test_sys, R2)  # optional -- phc
sage: test = get_solution_dicts(sol.output_file_contents,R2)  # optional -- phc
sage: str(sum([q[x1].real() for q in test]))[0:4]  # optional -- phc
'25.0'
```
sage.interfaces.phc.get_variable_list(output_file_contents)

Returns 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], [GJR2010], [GHJ2016], and [AGHJLPR2017].

```
class sage.interfaces.polymake.Polymake(script_subdirectory=None, logfile=None, server=None, server_tmpdir=None, seed=None, command=None)

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

Interface to the polymake interpreter.

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:

```
sage: p = polymake.rand_sphere(4, 20, seed=5)       # optional - polymake
sage: p                                             # optional - polymake
Random spherical polytope of dimension 4; seed=5...
sage: set_verbose(3)
```

```
sage: p.H_VECTOR                                     # optional - polymake
used package ppl
  The Parma Polyhedra Library (PPL): A C++ library for convex polyhedra
  and other numerical abstractions.
  http://www.cs.unipr.it/ppl/
  1 16 47 16 1
sage: set_verbose(0)
```

```
sage: p.F_VECTOR                                     # optional - polymake
used package ppl
  The Parma Polyhedra Library (PPL): A C++ library for convex polyhedra
  and other numerical abstractions.
  http://www.cs.unipr.it/ppl/
  20 101 162 81
```

```
sage: print(p.F_VECTOR._sage_doc_())                 # optional - polymake # 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]);
```
_eval_line (line, allow_use_file=True, wait_for_prompt=True, restart_if_needed=True, **kwds)
Evaluate a command.

INPUT:

• line, a command (string) to be evaluated
• allow_use_file (optional bool, default True), whether or not to use a file if the line is very long.
• wait_for_prompt (optional, default True), 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 (optional bool, default True), 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:

```
sage: p = polymake.cube(3)          # optional - polymake # 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 - polymake
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/

sage: set_verbose(0)
```

If polymake raises an error, the polymake interface raises a PolymakeError:

```
sage: polymake.eval('FOOBAR(3);')   # optional - polymake
Traceback (most recent call last):
  ...
PolymakeError: Undefined subroutine &Polymake::User::FOOBAR called...
```

If a command is incomplete, then polymake returns a continuation prompt. In that case, we raise an error:

```
sage: polymake.eval('print 3')      # optional - polymake
Traceback (most recent call last):
  ...
SyntaxError: Incomplete polymake command 'print 3'
sage: polymake.eval('print 3;')     # optional - polymake
'3'
```

However, if the command contains line breaks but eventually is complete, no error is raised:

```
sage: print(polymake.eval('$tmp="abc";\nprint $tmp;'))     # optional - polymake
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
doctest:warn

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
c.polymake.eval('print {}->F_VECTOR;'.format(c.name()), timeout=1)  # optional - polymake # not tested
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
cube of dimension 15
sage: c.N_VERTICES  # optional - polymake
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
g = polymake.new_object("Polytope", INEQUALITIES=[[5,-4,0,1],[-3,0,-4,1],[-2,1,0,0],[-4,4,1,-1],[0,0,1,0],[8,0,0,-1],[1,0,-1,0],[3,-1,0,0]])  # optional - polymake
g.hVECTOR  # optional - polymake
1 5 5 1
```
In the application ‘fan’, polymake can now compute the normal fan of \( q \) and its (primitive) rays:

\[
\begin{align*}
\text{sage: } & \text{polymake.application('fan')} & \text{# optional - polymake} \\
\text{sage: } & g = q.\text{normal_fan()} & \text{# optional - polymake} \\
\text{sage: } & g.\text{RAYS} & \text{# optional - polymake} \\
& -1 0 1/4 \\
& 0 -1 1/4 \\
& 1 0 0 \\
& 1 1 -1/4 \\
& 0 1 0 \\
& 0 0 -1 \\
& 0 -1 0 \\
& -1 0 0 \\
\text{sage: } & g.\text{RAYS.primitive()} & \text{# optional - polymake} \\
& -4 0 1 \\
& 0 -4 1 \\
& 1 0 0 \\
& 4 4 -1 \\
& 0 1 0 \\
& 0 0 -1 \\
& 0 -1 0 \\
& -1 0 0
\end{align*}
\]

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:**

\[
\begin{align*}
\text{sage: } & \text{polymake.console()} \\
\text{Traceback (most recent call last):} \\
... \\
\text{NotImplementedError: Please use polymake_console() function or the .} \\
\text{→ interact()} \text{ method}
\end{align*}
\]

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

**EXAMPLES:**

\[
\begin{align*}
\text{sage: } & \text{polymake.rand_sphere(4, 30, seed=15)} & \text{# optional - polymake} \\
\text{Random spherical polytope of dimension 4; seed=15...}
\end{align*}
\]

**get** *(cmd)*

Return the string representation of an object in the polymake interface.

**EXAMPLES:**
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
˓→polymake # random
doctest:warning
...
UserWarning: Polymake expects user interaction. We abort and return the options that Polymake provides.
There are 5 help topics matching 'TRIANGULATION':
1: objects/Visualization/Visual::Polytope/methods/TRIANGULATION
2: objects/Visualization/Visual::PointConfiguration/methods/TRIANGULATION
3: objects/Cone/properties/Triangulation and volume/TRIANGULATION
4: objects/PointConfiguration/properties/Triangulation and volume/
˓→TRIANGULATION
5: objects/Polytope/properties/Triangulation and volume/TRIANGULATION
```

If an unknown help topic is requested, a PolymakeError results:

```
sage: polymake.help('Triangulation')
# optional - 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:

```python
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
```

```python
sage: q.N_VERTICES # optional - polymake
4
```

```python
sage: q.BOUNDED # optional - polymake
1
```

```python
sage: q.VERITCES # optional - polymake
1 2 0 4
1 3 0 8
1 2 1 8
1 3 1 8
```

```python
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:**

```python
sage: c = polymake('cube(3)') # optional - polymake
```

```python
sage: d = polymake.cube(3) # optional - polymake
```

Equality is, for “big” objects such as polytopes, comparison by identity:

```python
sage: c == d # optional - polymake
False
```

However, the list of vertices is equal:

```python
sage: c.VERITCES == d.VERITCES # optional - polymake
True
```

**version()**

Version of the polymake installation.
EXAMPLES:

```python
sage: polymake.version() # optional - polymake
'3...
```

class sage.interfaces.polymake.PolymakeElement (parent, value, is_name=False, name=None)

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

Elements in the polymake interface.

EXAMPLES:

We support all “big” polymake types, Perl arrays of length different from one, and Perl scalars:

```python
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:

```python
sage: p.VERTEIS[2][2] # optional - polymake
-3319173990813887/4503599627370496
```

`bool()`

Return whether this polymake element is equal to `True`.

EXAMPLES:

```python
sage: from sage.interfaces.polymake import polymake
sage: polymake(0).bool() # optional polymake
False
sage: polymake(1).bool() # optional polymake
True
```

`full_typename()`

The name of the specialised type of this element.

EXAMPLES:

```python
sage: c = polymake.cube(4) # optional - polymake
sage: c.full_typename() # optional - polymake
'Polytope<Rational>'
sage: c.VERTEIS.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:

```python
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
20 101 162 81
```

However, explicit access is possible as well:

```
sage: p.get_member('F_VECTOR') # optional - polymake
20 101 162 81
```

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...]
```

Note that in the last example calling the erroneously constructed member function `type` still works:

```
sage: p.type() # optional - polymake
Polytope<Rational>[SAGE...]
```

`get_member_function(attrname)`

Request a member function of this element.

**NOTE:**

It is not checked whether a member function with the given name exists.

**EXAMPLES:**

```
sage: c = polymake.cube(2) # optional - polymake
sage: c.contains # optional - polymake
Member function 'contains' of Polymake::polytope::Polytope__Rational object
sage: V = polymake.new_object('Vector', [1,0,0]) # optional - polymake
sage: V # optional - polymake
1 0 0
sage: c.contains(V) # optional - polymake
1
```

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" at input line 1.
```
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',
 'CONE_AMBIENT_DIM',
 'CONE_DIM',
...
 'VERTICES_IN_FACETS']
sage: c.list_properties()  # optional - polymake
CONE_AMBIENT_DIM, CONE_DIM, FACETS, AFFINE_HULL, VERTICES_IN_FACETS, BOUNDED...
```

A computation can change the list of known properties:

```
sage: c.F_VECTOR  # optional - polymake
16 32 24 8
sage: c.known_properties()  # optional - polymake
['AFFINE_HULL',
 'BOUNDED',
 'COMBINATORIAL_DIM',
 'CONE_AMBIENT_DIM',
 'CONE_DIM',
 'DUAL_H_VECTOR',
 'FACETS',
 'FAR_FACE',
 'FEASIBLE',
 'FULL_DIM',
 'F_VECTOR',
 'GRAPH',
 'LINEALITY_DIM',
 'LINEALITY_SPACE',
 'N_FACETS',
 'N_VERTICES',
 'POINTED',
 'SIMPLE',
 'SIMPLICIAL',
 'VERTICES',
 '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>'
```

(continues on next page)
### `typename()`

The name of the underlying base type of this element in polymake.

**EXAMPLES:**

```python
sage: c = polymake.cube(4)  # optional - polymake
sage: c.typename()         # optional - polymake
'Polytope'
```

### `typeof()`

Returns the type of a polymake “big” object, and its underlying Perl type.

**NOTE:**

This is mainly for internal use.

**EXAMPLES:**

```python
sage: p = polymake.rand_sphere(3, 13, seed=12)  # optional - polymake
sage: p.typeof()                                 # optional - polymake
('Polymake::polytope::Polytope__Rational', 'ARRAY')
```

On “small” objects, it just returns empty strings:

```python
sage: p.N_VERTICES.typeof()                      # optional - polymake
('', '')
```

### `exception` `sage.interfaces.polymake.PolymakeError`  
**Bases:** `exceptions.RuntimeError`

Raised if polymake yields an error message.

### `class` `sage.interfaces.polymake.PolymakeFunctionElement`  
**Bases:** `sage.interfaces.expect.FunctionElement`

A callable (function or member function) bound to a polymake element.

**EXAMPLES:**

```python
sage: c = polymake.cube(2)  # optional - polymake
sage: V = polymake.new_object('Vector', [1,0,0])  # optional - polymake
```
sage.interfaces.polymake.polymake_console(command="")
Spin 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()
Returns the polymake interface object defined in sage.interfaces.polymake.

EXAMPLES:

sage: from sage.interfaces.polymake import reduce_load_Polymake
sage: reduce_load_Polymake()
Polymake
POV-Ray, THE PERSISTENCE OF VISION RAY TRACER

**class** sage.interfaces.povray.POVRay

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()**
PARALLEL INTERFACE TO THE SAGE INTERPRETER

This is an expect interface to 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:

\[
\text{sage: } v = [\text{PSage() for } _\_ \text{ in range(3)}]
\]

Next, request factorization of one random integer in each copy.

\[
\text{sage: } w = [x('factor(2^%s-1)'% randint(250,310)) \text{ for } x \text{ in } v] \quad \# \text{ long time (5s on sage. math, 2011)}
\]

Print the status:

\[
\text{sage: } w \quad \# \text{ long time, random output (depends on timing)}
\]

\[
[3 \times 11 \times 31^2 \times 311 \times 11161 \times 11471 \times 73471 \times 715827883 \times 2147483647 \times 4649919401 \times \ldots
\]

\[
\ldots 18158209813151 \times 5947603221397891 \times 29126056043168521,
\]

\[
\langle\text{currently executing code}>,
\]

\[
9623 \times 68492481833 \times \ldots
\]

\[
23579543011798993222850893929565870383844167873851502677311057483194673
\]

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:

\[
\text{sage: } w \quad \# \text{ long time, random output}
\]

\[
[3 \times 11 \times 31^2 \times 311 \times 11161 \times 11471 \times 73471 \times 715827883 \times 2147483647 \times 4649919401 \times \ldots
\]

\[
\ldots 18158209813151 \times 5947603221397891 \times 29126056043168521,
\]

\[
23^2 \times 47 \times 89 \times 178481 \times 4103188409 \times 199957736328435366769577 \times \ldots
\]

\[
44667711762797798403039426178361,
\]

\[
9623 \times 68492481833 \times \ldots
\]

\[
23579543011798993222850893929565870383844167873851502677311057483194673
\]

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()
CHAPTER
THIRTY-SIX

INTERFACE TO QEPCAD

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 \texttt{qepcad_formula} which helps construct quantified formulas in the syntax QEPCAD requires.

```sage
sage: var('a,b,c,d,x,y,z')
(a, b, c, d, x, y, z)
sage: qf = qepcad_formula
```

We start with a simple example. Consider an arbitrarily-selected ellipse:

```sage
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.
Then we run qepcad to get the answer:

```
sage: qepcad(F)                               # optional - qepcad
8 x^2 - 8 x - 29 <= 0
```

How about the projection onto the $y$ axis?

```
sage: qepcad(qf.exists(x, ellipse == 0))      # optional - qepcad
8 y^2 + 16 y - 85 <= 0
```

QEPCAD deals with more quantifiers than just ‘exists’, of course. Besides the standard ‘forall’, there are also ‘for infinitely many’, ‘for all but finitely many’, ‘for a connected subset’, and ‘for exactly $k$’. The qepcad() documentation has examples of all of these; here we will just give one example.

First we construct a circle:

```
sage: circle = x^2 + y^2 - 3
```

For what values $k$ does a vertical line $x = k$ intersect the combined figure of the circle and ellipse exactly three times?

```
sage: F = qf.exactly_k(3, y, circle * ellipse == 0); F
(3 x^2 + 2 x y + y^2 - x + y - 7) (x^2 + y^2 - 3) = 0
```

```
sage: qepcad(F)                               # not tested (random order)
x^2 - 3 <= 0 /
8 x^2 - 8 x - 29 <= 0 /
8 x^4 - 26 x^2 - 4 x + 13 >= 0 /
8 x^4 - 26 x^2 - 4 x + 13 = 0 /
x^2 - 3 = 0 /
8 x^2 - 8 x - 29 = 0
```

Here we see that the solutions are among the eight $(4 + 2 + 2)$ roots of the three polynomials inside the brackets, but not all of these roots are solutions; the polynomial inequalities outside the brackets are needed to select those roots that are solutions.

QEPCAD also supports an extended formula language, where $\_\text{root}_k P(x, y)$ refers to a particular zero of $P(x, y)$ (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.

```
sage: qepcad(F, solution='geometric')         # not tested (random order)
x = $\_\text{root}_1$ 8 x^2 - 8 x - 29
\/
8 x^4 - 26 x^2 - 4 x + 13 = 0
\/
x = $\_\text{root}_{-1}$ x^2 - 3
```

We then see that the 6 solutions correspond to the vertical tangent on the left side of the ellipse, the four intersections between the ellipse and the circle, and the vertical tangent on the right side of the circle.

Let us do some basic formula simplification and visualization. We will look at the region which is inside both the ellipse and the circle:

```
sage: F = qf.and_(ellipse < 0, circle < 0); F
[3 x^2 + 2 x y + y^2 - x + y - 7 < 0 /
x^2 + y^2 - 3 < 0]
sage: qepcad(F)                               # not tested (random order)
y^2 + 2 x y + y + 3 x^2 - x - 7 < 0 /
y^2 + x^2 - 3 < 0
```

248 Chapter 36. Interface to QEPCAD
We get back the same formula we put in. This is not surprising (we started with a pretty simple formula, after all), but it is not very enlightening either. Again, if we ask for a ‘geometric’ output, then we see an output that lets us understand something about the shape of the solution set.

```sage
def qepcad(F, solution='geometric'):
    # not tested (random order)
    return [{}]
```

```markdown
\[
\begin{align*}
\text{sage: } & \text{qepcad(F, solution='geometric')} \\
& \quad \# \text{ not tested (random order)} \\
& \left[ \\
& \quad \left[ \\
& \quad \text{x} = \text{\_root\_2} 8 \text{x^4} - 26 \text{x^2} - 4 \text{x} + 13 \\
& \quad \text{\_root\_2} 8 \text{x^4} - 26 \text{x^2} - 4 \text{x} + 13 \\
& \quad 8 \text{x^4} - 26 \text{x^2} - 4 \text{x} + 13 < 0 \\
& \quad \right] \\
& \quad \backslash \\
& \quad \text{y^2} + 2 \text{x} \text{y} + \text{y} + 3 \text{x^2} - \text{x} - 7 < 0 \\
& \quad \backslash \\
& \quad \text{y^2} + \text{x^2} - 3 < 0 \\
& \quad \right] \\
& \backslash \\
& \left[ \\
& \quad \text{x} > \text{\_root\_2} 8 \text{x^4} - 26 \text{x^2} - 4 \text{x} + 13 \\
& \quad \text{\_root\_2} 8 \text{x^4} - 26 \text{x^2} - 4 \text{x} + 13 \\
& \quad \text{y^2} + \text{x^2} - 3 < 0 \\
& \quad \right]
\end{align*}
\]
```

There is another reason to prefer output using _root_ expressions; not only does it sometimes give added insight into the geometric structure, it also can be more efficient to construct. Consider this formula for the projection of a particular semicircle onto the \(x\) axis:

```sage
def qepcad(F, solution='extended'):
    # not tested (random order)
    return [{}]
```

```markdown
\[
\begin{align*}
\text{sage: } & \text{qepcad(F, solution='extended')} \\
& \quad \# \text{ not tested (random order)} \\
& \text{x^2} - 3 \leq 0 \quad \backslash \quad [ \text{x} > 0 \quad \backslash \quad 2 \text{x^2} - 3 < 0 ]
\end{align*}
\]
```

Here, the formula \(x > 0\) had to be introduced in order to get a solution formula; the original CAD of \(F\) did not include the polynomial \(x\). To avoid having QEPCAD do the extra work to come up with a solution formula, we can tell it to use the extended language; it is always possible to construct a solution formula in the extended language without introducing new polynomials.

```sage
def qepcad(F, solution='extended'):
    # not tested (random order)
    return [{}]
```

```markdown
\[
\begin{align*}
\text{sage: } & \text{qepcad(F, solution='extended')} \\
& \quad \# \text{ not tested (random order)} \\
& \text{x^2} - 3 \leq 0 \quad \backslash \quad [ \text{x} > \text{\_root\_1} 2 \text{x^2} - 3
\end{align*}
\]
```

Up to this point, all the output we have seen has basically been in the form of strings; there is no support (yet) for parsing these outputs back into sage polynomials (partly because sage does not yet have support for symbolic conjunctions and disjunctions). The function `qepcad()` supports three more output types that give numbers which can be manipulated in sage: any-point, all-points, and cell-points.

These output types give dictionaries mapping variable names to values. With any-point, `qepcad()` either produces a single dictionary specifying a point where the formula is true, or raises an exception if the formula is false everywhere. With all-points, `qepcad()` either produces a list of dictionaries for all points where the formula is true, or raises an exception if the formula is true on infinitely many points. With cell-points, `qepcad()` produces a list of dictionaries with one point for each cell where the formula is true. (This means you will have at least one point in each connected component of the solution, although you will often have many more points than that.)
Let us revisit some of the above examples and get some points to play with. We will start by finding a point on our ellipse.

```python
sage: p = qepcad(ellipse == 0, solution='any-point'); p
# optional - qepcad
{'x': -1.468501968502953?, 'y': 0.9685019685029527?}
```

(Note that despite the decimal printing and the question marks, these are really exact numbers.)

We can verify that this point is a solution. To do so, we create a copy of ellipse as a polynomial over $\mathbb{Q}$ (instead of a symbolic expression).

```python
sage: pellipse = QQ['x,y'](ellipse)
sage: pellipse(**p) == 0
# optional - qepcad
True
```

For cell-points, let us look at points not on the ellipse.

```python
sage: pts = qepcad(ellipse != 0, solution='cell-points'); pts
# optional - qepcad
[{'x': 4, 'y': 0},
 {'x': 2.468501968502953?, 'y': 1},
 {'x': 2.468501968502953?, 'y': -9},
 {'x': 1/2, 'y': 9},
 {'x': 1/2, 'y': -1},
 {'x': 1/2, 'y': -5},
 {'x': -1.468501968502953?, 'y': 3},
 {'x': -1.468501968502953?, 'y': -1},
 {'x': -3, 'y': 0}]
```

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]
# optional - qepcad
[True, True, True, True, True, True, True, True, True]
```

Finally, for all-points, let us look again at finding vertical lines that intersect the union of the circle and the ellipse exactly three times.

```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: pts = qepcad(F, solution='all-points'); pts
# optional - qepcad
[{'x': 1.732050807568878?}, {'x': 1.731054913462534?}, {'x': 0.678911384208004?}, {'x': -0.9417727377417167?}, {'x': -1.468193559928821?}, {'x': -1.468501968502953?}]
```

Since $y$ is bound by the quantifier, the solutions only refer to $x$.

We can substitute one of these solutions into the original equation:

```python
sage: pt = pts[0]
# optional - qepcad
sage: pcombo = QQ['x,y'](circle * ellipse)
sage: intersections = pcombo(y=polygen(AA, 'y'), **pt); intersections
# optional - qepcad
y^4 + 4.464101615137755?*y^3 + 0.2679491924311227?*y^2
```

and verify that it does have three roots:
Let us check all six solutions.

```
sage: [len(pcombo(y=polygen(AA, 'y'), **p).roots()) for p in pts]  # optional - qepcad
[3, 3, 3, 3, 3, 3]
```

We said earlier that we can run QEPCAD either automatically or interactively. Now that we have discussed the automatic modes, let us turn to interactive uses.

If the `qepcad()` function is passed `interact=True`, then instead of returning a result, it returns an object of class `Qepcad` representing a running instance of QEPCAD that you can interact with. For example:

```
sage: qe = qepcad(qf.forall(x, x^2 + b*x + c > 0), interact=True); qe  # optional - qepcad
QEPCAD object in phase 'Before Normalization'
```

This object is a fairly thin wrapper over QEPCAD; most QEPCAD commands are available as methods on the `Qepcad` object. Given a `Qepcad` object `qe`, you can type `qe.[tab]` to see the available QEPCAD commands; to see the documentation for an individual QEPCAD command, for example `d_setting`, you can type `qe.d_setting?`.

(In QEPCAD, this command is called `d-setting`. We systematically replace hyphens with underscores for this interface.)

The execution of QEPCAD is divided into four phases. Most commands are not available during all phases. We saw above that QEPCAD starts out in phase 'Before Normalization'. We see that the `d_cell` command is not available in this phase:

```
sage: qe.d_cell()  # optional - qepcad
Error GETCID: This command is not active here.
```

We will focus here on the fourth (and last) phase, 'Before Solution', because this interface has special support for some operations in this phase. Consult the QEPCAD documentation for information on the other phases.

We can tell QEPCAD to finish off the current phase and move to the next with its `go` command. (There is also the `step` command, which partially completes a phase for phases that have multiple steps, and the `finish` command, which runs QEPCAD to completion.)

```
sage: qe.go()  # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (x)'
sage: qe.go()  # optional - qepcad
QEPCAD object has moved to phase 'Before Choice'
sage: qe.go()  # optional - qepcad
QEPCAD object has moved to phase 'Before Solution'
```

Note that the `Qepcad` object returns the new phase whenever the phase changes, as a convenience for interactive use; except that when the new phase is 'EXITED', the solution formula printed by QEPCAD is returned instead.

```
sage: qe.go()  # optional - qepcad
4 c - b^2 > 0
sage: qe  # optional - qepcad
QEPCAD object in phase 'EXITED'
```

Let us pick a nice, simple example, return to phase 4, and explore the resulting `qe` object.
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: 
\[
\begin{align*}
    x &< -\sqrt{3}, \\
    x &> \sqrt{3}, \\
    -\sqrt{3} &< x < \sqrt{3}, \\
    x &= \sqrt{3}, \\
    x &= -\sqrt{3}.
\end{align*}
\]
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: 
\[
\begin{align*}
    y &< 0, \\
    y &> 0, \\
    y &= 0.
\end{align*}
\]
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 \texttt{d\_cell}. For instance, let us look at the cell for the upper semicircle:

\[
\texttt{sage: qe\.d\_cell(3, 4) \hfill \# optional - qepcad}
\]

\[
\begin{tabular}{ll}
Level & 2 \\
Dimension & 1 \\
Number of children & 0 \\
Truth value & T by trial evaluation. \\
Degrees after substitution & Not known yet or No polynomial. \\
Multiplicities & ((1,1)) \\
Signs of Projection Factors & \begin{align*}
    \text{Level 1} & : (-) \\
    \text{Level 2} & : (0)
\end{align*} \\
Sample point & \begin{align*}
    \text{The sample point is in a PRIMITIVE representation.} \\
    \text{alpha} & = \text{the unique root of } x^2 - 3 \text{ between 0 and 4} \\
    & = 1.7320508076- \\
    \text{Coordinate 1} & = 0 \\
    & = 0.0000000000 \\
    \text{Coordinate 2} & = \text{alpha} \\
    & = 1.7320508076-
\end{align*}
\end{tabular}
\]

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 \texttt{QepcadCell}. These cells can be created with the \texttt{cell()} method; for example:

\[
\texttt{sage: c = qe.cell(3, 4) \hfill \# optional - qepcad}
\]
A `QepcadCell` has accessor methods for the important state held within a cell. For instance:

```python
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.

```python
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.

```python
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.)

```python
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:

```python
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:

```python
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:

```python
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.)

```python
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 $R^1$ such that the stack over $c$ contains exactly two cells on the ellipse, and also contains exactly two cells on the circle.

Our input polynomials are ‘level-2 projection factors’, we see:

```python
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:

```python
sage: qe.cell(12,2).signs()[1][0]  # optional - qepcad
0
```

So we can update the truth values as desired like this:

```python
sage: for c in qe.cell():  # optional - qepcad
  ....: count_ellipse = 0
  ....: count_circle = 0
  ....: for c2 in c:
  ....:     count_ellipse += (c2.signs()[1][0] == 0)
  ....:     count_circle += (c2.signs()[1][1] == 0)
  ....:     c.set_truth(count_ellipse == 2 and count_circle == 2)
```

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.)

```python
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)

The wrapper for QEPCAD.

answer()  
For a QEPCAD instance which is finished, return the simplified quantifier-free formula that it printed just before exiting.

EXAMPLES:

sage: qe = qepcad(x^3 - x == 0, interact=True)  # optional - qepcad
sage: qe.finish()  # not tested (random order)
x - 1 <= 0 \( x + 1 >= 0 \) \( x = 0 \) \( x - 1 = 0 \) \( x + 1 = 0 \)
sage: qe.answer()  # not tested (random order)
x - 1 <= 0 \( x + 1 >= 0 \) \( x = 0 \) \( x - 1 = 0 \) \( x + 1 = 0 \)

assume(assume)  
The following documentation is from qepcad.help.  
Add an assumption to the problem. These will not be included in the solution formula.

For example, with input \((\exists x)[ a x^2 + b x + c = 0]\), if we issue the command

\[
\text{assume } [ a /= 0 ]
\]

we will get the solution formula \( b^2 - 4 a c >= 0 \). Without the assumption we’d get something like \([a = 0 \land b /= 0]\) / \([a /= 0 \land 4 a c - b^2 <= 0]\) / \([a = 0 \land b = 0 \land c = 0]\).

EXAMPLES:

sage: var('a,b,c,x')
(a, b, c, x)
sage: qf = qepcad_formula
sage: qe = qepcad(qf.exists(x, a*x^2 + b*x + c == 0), interact=True)  # optional - qepcad
sage: qe.assume(a != 0)  # optional - qepcad
sage: qe.finish()  # optional - qepcad
4 a c - b^2 <= 0


cell(*index)  
Given a cell index, returns a QepcadCell wrapper for that cell. Uses a cache for efficiency.

EXAMPLES:

sage: qe = qepcad(x + 3 == 42, interact=True)  # optional - qepcad
sage: qe.go(); qe.go(); qe.go()  # optional - qepcad
QEPCAD object has moved to phase 'At the end of projection phase'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.cell(2)  # optional - qepcad
QEPCAD cell (2)
sage: qe.cell(2) is qe.cell(2)  # optional - qepcad
True

final_stats()  
For a QEPCAD instance which is finished, return the statistics that it printed just before exiting.

EXAMPLES:

sage: qe = qepcad(x == 0, interact=True)  # optional - qepcad
sage: qe.finish()  # optional - qepcad

(continues on next page)
x = 0

```python
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
'Before Normalization'
sage: qe.go()  # optional - qepcad
QEPCAD object has moved to phase 'At the end of projection phase'
sage: qe.phase()  # optional - qepcad
'At the end of projection phase'
sage: qe.go()  # optional - qepcad
QEPCAD object has moved to phase 'Before Choice'
sage: qe.phase()  # optional - qepcad
'Before Choice'
sage: qe.go()  # optional - qepcad
QEPCAD object has moved to phase 'Before Solution'
sage: qe.phase()  # optional - qepcad
'Before Solution'
sage: qe.go()  # optional - qepcad
3 x - 2 > 0
sage: qe.phase()  # optional - qepcad
'EXITED'
```

### set_truth_value(index, nv)

Given a cell index (or a cell) and an integer, set the truth value of the cell to that integer.

Valid integers are 0 (false), 1 (true), and 2 (undetermined).

**EXAMPLES:**

```python
```
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:

```python
sage: var('x,y')
(x, y)
sage: qf = qepcad_formula
sage: qe = qepcad(qf.and_(x^2 + y^2 - 3 == 0, x + y > 0), interact=True) #
˓→optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.solution_extension('E') # not tested (random
˓→order)
x > _root_1 2 x^2 - 3 /\ y^2 + x^2 - 3 = 0 /\ [ 2 x^2 - 3 > 0 \ / y = _root_-1,
˓→ y^2 + x^2 - 3 ]
sage: qe.solution_extension('G') # not tested (random
˓→order)
[ [ 2 x^2 - 3 < 0
\ / x = _root_-1 2 x^2 - 3
] /\ y = _root_-1 y^2 + x^2 - 3 ]
] /\ x^2 - 3 <= 0
\ / x > _root_-1 2 x^2 - 3
\ / y^2 + x^2 - 3 = 0
sage: qe.solution_extension('T') # not tested (random
˓→order)
y + x > 0 /\ y^2 + x^2 - 3 = 0
```

class sage.interfaces.qepcad.QepcadCell (parent, lines)
A wrapper for a QEPCAD cell.

index() 
Give the index of a QEPCAD cell.
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.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:**

```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.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:**

```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.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'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.cell(4).sample_point_dict() # optional - qepcad
{'x': 1.618033988749895?}
```

**set_truth(v)**

Set the truth value of this cell, as used by QEPCAD for solution formula construction.

The argument v should be either a boolean or None (which will set the truth value to 'undetermined').

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: qe = qepcad(x^2 + y^2 == 1, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.solution_extension('T') # optional - qepcad
y^2 + x^2 - 1 = 0
sage: qe.cell(3, 3).set_truth(True) # optional - qepcad
sage: qe.solution_extension('T') # optional - qepcad
y^2 + x^2 - 1 <= 0
```

**signs()**

Return the sign vector of a QEPCAD cell.

This is a list of lists. The outer list contains one element for each level of the cell; the inner list contains one element for each projection factor at that level. These elements are either -1, 0, or 1.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: qe = qepcad(x^2 + y^2 == 1, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: 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
```

(continues on next page)
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
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 (memcells=None, maxread=None, log-
file=None, server=None)
Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.
expect.Expect
The low-level wrapper for QEPCAD.
sage.interfaces.qepcad.qepcad (formula, assume=None, interact=False, solution=None,
vars=None, **kwargs)
Quantifier elimination and formula simplification using QEPCAD B.

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_
For which values of $a$, $b$, $c$ does $ax^2 + bx + c$ have 2 real zeroes?

```
sage: exact2 = qepcad(qf.exactly_k(2, x, a*x^2 + b*x + c == 0)); exact2  # not tested (random order)
a /= 0 /\ 4 a c - b^2 < 0
```

one real zero?

```
sage: exact1 = qepcad(qf.exactly_k(1, x, a*x^2 + b*x + c == 0)); exact1  # not tested (random order)
[ a > 0 /\ 4 a c - b^2 = 0 ] /\ [ a < 0 /\ 4 a c - b^2 = 0 ] /\ [ a = 0 /\ 4 a c - b^2 < 0 ]
```

No real zeroes?

```
sage: exact0 = qepcad(qf.forall(x, a*x^2 + b*x + c != 0)); exact0  # not tested (random order)
4 a c - b^2 >= 0 /\ c /= 0 /\ [ b = 0 /\ 4 a c - b^2 > 0 ]
```

$3^{75}$ real zeroes?

```
sage: qepcad(qf.exactly_k(3^75, x, a*x^2 + b*x + c == 0))  # optional - qepcad
FALSE
```

We can check that the results don’t overlap:

```
sage: qepcad(r'[^\$\s] /\ [^\$\s])' % (exact0, exact1), vars='a,b,c')  # not tested (random order)
FALSE
sage: qepcad(r'[^\$\s] /\ [^\$\s])' % (exact0, exact2), vars='a,b,c')  # not tested (random order)
FALSE
sage: qepcad(r'[^\$\s] /\ [^\$\s])' % (exact1, exact2), vars='a,b,c')  # not tested (random order)
FALSE
```

and that the union of the results is as expected:

```
sage: qepcad(r'[^\$\s] /\ [^\$\s] /\ [^\$\s])' % (exact0, exact1, exact2), vars=(a,b,c))
  # not tested (random order)
b /= 0 /\ a /= 0 /\ c /= 0
```

So we have finitely many zeroes if $a$, $b$, or $c$ is nonzero; which means we should have infinitely many zeroes if they are all zero.
The polynomial is nonzero almost everywhere iff it is not identically zero.

```
sage: qepcad(qf.infinitely_many(x, a*x^2 + b*x + c == 0))  # not tested
⇒ (random order)
a = 0 \∧ b = 0 \∧ c = 0
```

The non-zeros are continuous iff there are no zeroes or if the polynomial is zero.

```
sage: qepcad(qf.all_but_finitely_many(x, a*x^2 + b*x + c != 0))  # not tested
⇒ (random order)
4 a^2 - b^2 >= 0 \∧ [ a = 0 \∧ 4 a c - b^2 > 0 ]
```

The zeroes are continuous iff there are no or one zeroes, or if the polynomial is zero:

```
sage: qepcad(qf.connected_subset(x, a*x^2 + b*x + c != 0))  # not tested
⇒ (random order)
a = 0 \∧ 4 a c - b^2 = 0
```

Since polynomials are continuous and \( y > 0 \) is an open set, they are positive infinitely often iff they are positive at least once.

```
sage: qepcad(qf.infinitely_many(x, a*x^2 + b*x + c > 0))  # not tested
⇒ (random order)
c > 0 \∧ a > 0 \∧ 4 a c - b^2 < 0
```

However, since \( y >= 0 \) is not open, the equivalence does not hold if you replace ‘positive’ with ‘nonnegative’. (We assume \( a \neq 0 \) to get simpler formulas.)

```
sage: qepcad(qf.infinitely_many(x, a*x^2 + b*x + c >= 0), assume=(a != 0))  # not tested
⇒ (random order)
a > 0 \∧ 4 a c - b^2 < 0
```

```
sage: from sage.interfaces.qepcad import qepcad
sage: qepcad_banner()  # optional - qepcad
 butcher to QEPCAD startup banner.

EXAMPLES:

```
sage: from sage.interfaces.qepcad import qepcad
sage: qepcad_banner()  # optional - qepcad
```
sage.interfaces.qepcad.qepcad_console(memcells=None)
Run QEPCAD directly. To exit early, press Control-C.

EXAMPLES:

```python
sage: qepcad_console() # not tested
...
Enter an informal description between '[:' and ':]:'
```

class sage.interfaces.qepcad.qepcad_formula_factory
Contains routines to help construct formulas in QEPCAD syntax.

A (v, formula)
Given a variable (or list of variables) and a formula, returns the universal quantification of the formula over
the variables.

This method is available both as `forall()` and `A()` (the QEPCAD name for this quantifier).

The input formula may be a `qformula` as returned by the methods of `qepcad_formula`, a symbolic
equality or inequality, or a polynomial \( p \) (meaning \( p = 0 \)).

EXAMPLES:

```python
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 quanti-
fier).

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:
Given a variable (or list of variables) and a formula, returns the existential quantification of the formula over the variables.

This method is available both as `exists()` and `E()` (the QEPCAD name for this quantifier).

The input formula may be a `qformula` as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial \( p \) (meaning \( p = 0 \)).

**EXAMPLES:**

```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]
```

**E (v, formula)**

Given a variable and a formula, returns a new formula which is true iff the original formula was true for infinitely many values of the variable.

This method is available both as `infinitely_many()` and `F()` (the QEPCAD name for this quantifier).

The input formula may be a `qformula` as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial \( p \) (meaning \( p = 0 \)).

**EXAMPLES:**

```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.F(b, b^2 == a)
(F b)[b^2 == a]
```

**F (v, formula)**

Given a variable and a formula, returns a new formula which is true iff the original formula was true for all but finitely many values of the variable.

This method is available both as `all_but_finitely_many()` and `G()` (the QEPCAD name for this quantifier).

The input formula may be a `qformula` as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial \( p \) (meaning \( p = 0 \)).

**EXAMPLES:**

```python
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.all_but_finitely_many(a, a^2 + b > b^2 + a)
(F a)[a^2 + b > b^2 + a]
sage: qf.F(b, b^2 != a)
(F b)[b^2 /= a]
```

**G (v, formula)**

Given a variable and a formula, returns a new formula which is true iff the original formula was true for all but finitely many values of the variable.

This method is available both as `all_but_finitely_many()` and `G()` (the QEPCAD name for this quantifier).

The input formula may be a `qformula` as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial \( p \) (meaning \( p = 0 \)).

**EXAMPLES:**
X(k, v, formula, allow_multi=False)
Given a nonnegative integer k, a variable, and a formula, returns a new formula which is true iff the original formula is true for exactly k values of the variable.

This method is available both as exactly_k() and X() (the QEPCAD name for this quantifier).

(Note that QEPCAD does not support k = 0 with this syntax, so if k = 0 is requested we implement it with forall() and not().)

The input formula may be a qformula as returned by the methods of qepcad_formula, a symbolic equality or inequality, or a polynomial p (meaning p = 0).

EXAMPLES:

sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.exactly_k(1, x, x^2 + a*x + b == 0)
(X1 x)[a x + x^2 + b = 0]
sage: qf.exactly_k(0, b, a*b == 1)
(A b)[~a b = 1]

all_but_finitely_many(v, formula)
Given a variable and a formula, returns a new formula which is true iff the original formula was true for all but finitely many values of the variable.

This method is available both as all_but_finitely_many() and G() (the QEPCAD name for this quantifier).

The input formula may be a qformula as returned by the methods of qepcad_formula, a symbolic equality or inequality, or a polynomial p (meaning p = 0).

EXAMPLES:

sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.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]

and_(*formulas)
Return the conjunction of its input formulas.

(This method would be named ‘and’ if that were not a Python keyword.)

Each input formula may be a qformula as returned by the methods of qepcad_formula, a symbolic equality or inequality, or a polynomial p (meaning p = 0).

EXAMPLES:
sage: var('a,b,c,x')
(a, b, c, x)
sage: qf = qepcad_formula
sage: qf.and_(a+b, a+c, b+c != 0)
(a b = 0 \\ a c = 0 \\ \ b c /= 0]
sage: qf.and_(a*x^2 == 3, qf.or_(a > b, b > c))
[a x^2 = 3 \\ [a > b \\ / b > c]]

atomic (lhs, op='=', rhs=0)
Construct a QEPCAD formula from the given inputs.

INPUT:

- lhs – a polynomial, or a symbolic equality or inequality
- op – a relational operator, default ‘=’
- rhs – a polynomial, default 0

If lhs is a symbolic equality or inequality, then op and rhs are ignored.

This method works by printing the given inequality, 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:**

```python
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.exactly_k(1, x, x^2 + a*x + b == 0)
(X1 x)[a x + x^2 + b = 0]
sage: qf.exactly_k(0, b, a*b == 1)
(A b)[~a b = 1]
```

**exists** \((v, formula)\)

Given a variable (or list of variables) and a formula, returns the existential quantification of the formula over the variables.

This method is available both as `exists()` and `E()` (the QEPCAD name for this quantifier).

The input formula may be a `qformula` as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial \(p\) (meaning \(p = 0\)).

**EXAMPLES:**

```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]
```

**forall** \((v, formula)\)

Given a variable (or list of variables) and a formula, returns the universal quantification of the formula over the variables.

This method is available both as `forall()` and `A()` (the QEPCAD name for this quantifier).

The input formula may be a `qformula` as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial \(p\) (meaning \(p = 0\)).

**EXAMPLES:**

```python
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]
```
Constructs a QEPCAD formula from the given input.

**INPUT:**

1. `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:**

```python
sage: var('a,b,c,x')
(a, b, c, x)
sage: qf = qepcad_formula
sage: qf.formula(a*x + b)
a x + b = 0
sage: qf.formula((a*x^2 + b*x + c, a != 0))
[a x^2 + b x + c = 0 \&\& a /= 0]
```

Return the equivalence of its input formulas (that is, given formulas \( P \) and \( Q \), returns ‘\( P \iff Q \)’.

The input formulas may be a `qformula` as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial \( p \) (meaning \( p = 0 \)).

**EXAMPLES:**

```python
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]
```

Return the implication of its input formulas (that is, given formulas \( P \) and \( Q \), returns ‘\( P \implies Q \)’.

The input formulas may be a `qformula` as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial \( p \) (meaning \( p = 0 \)).

**EXAMPLES:**

```python
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]
```

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.

**EXAMPLES:**

```python
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]
```
This method is available both as `infinitely_many()` and `F()` (the QEPCAD name for this quantifier).

The input formula may be a `qformula` as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial $p$ (meaning $p = 0$).

EXAMPLES:

```
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.infinitely_many(a, a^2 + b > b^2 + a)
(F a)[a^2 + b > b^2 + a]
sage: qf.F(b, b^2 != a)
(F b)[b^2 /= a]
```

`not_(formula)`

Return the negation of its input formula.

(This method would be named ‘not’ if that were not a Python keyword.)

The input formula may be a `qformula` as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial $p$ (meaning $p = 0$).

EXAMPLES:

```
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.not_(a > b)
[^a > b]
sage: qf.not_(a^2 + b^2)
[^a^2 + b^2 = 0]
sage: qf.not_(qf.and_(a > 0, b < 0))
[^{a > 0 \land b < 0}]
```

`or_(*formulas)`

Return the disjunction of its input formulas.

(This method would be named ‘or’ if that were not a Python keyword.)

Each input formula may be a `qformula` as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial $p$ (meaning $p = 0$).

EXAMPLES:

```
sage: var('a,b,c,x')
(a, b, c, x)
sage: qf = qepcad_formula
sage: qf.or_(a*b, a*c, b*c != 0)
[a b = 0 \lor a c = 0 \lor b c /= 0]
sage: qf.or_(a*x^2 == 3, qf.and_(a > b, b > c))
[a x^2 = 3 \lor [a > b \land b > c]]
```

`quantifier(kind, v, formula, allow_multi=True)`

A helper method for building quantified QEPCAD formulas; not expected to be called directly.

Takes the quantifier kind (the string label of this quantifier), a variable or list of variables, and a formula, and returns the quantified formula.

EXAMPLES:
\begin{verbatim}
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.quantifier('NOT_A_REAL_QEPCAD_QUANTIFIER', a, a*b==0)
(NOT_A_REAL_QEPCAD_QUANTIFIER a)[a b = 0]
sage: qf.quantifier('FOO', (a, b), a*b)
(FOO a)(FOO b)[a b = 0]
\end{verbatim}

\texttt{sage.interfaces.qepcad.qepcad_version()}

Return a string containing the current QEPCAD version number.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: qepcad_version() # random, optional - qepcad
'Version B 1.69, 16 Mar 2012'
\end{verbatim}

\texttt{class sage.interfaces.qepcad.qformula(formula, vars, qvars=[])}

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
[10000000000000000051, 100000000000000000039]
sage: t
'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)
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
Proper factors so far: []
sage: q
Proper factors so far: [100000000000000000039, 1000000000000000000117]
sage: q
[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:
\texttt{sage: n = next\_prime(2^{310})*next\_prime(2^{300})}
\texttt{sage: qs = qsieve(n, block=False)}
\texttt{sage: qs}
Proper factors so far: []
\texttt{sage: qs.quit()}
\texttt{sage: qs}
Factorization was terminated early.

\textbf{time}()
Return the time in seconds (as a string) that it took to factor \textit{n}, or return ‘?’ if the factorization has not completed or the time is unknown.
CHAPTER
THIRTEEN

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
[1] 0.09615385 0.17857143 0.32258065 0.15625000 0.04608295
sage: 1/x
[1] 0.09615385 0.17857143 0.32258065 0.15625000 0.04608295
```

The following assignment creates a vector $y$ with 11 entries which consists of two copies of $x$ with a 0 in between:

```
sage: y = r([x,0,x]); y
[1] 10.4 5.6 3.1 6.4 21.7 0.0 10.4 5.6 3.1 6.4 21.7
```

Vector Arithmetic

The following command generates a new vector $v$ of length 11 constructed by adding together (element by element) $2x$ repeated 2.2 times, $y$ repeated just once, and 1 repeated 11 times:

```
sage: v = 2*x+y+1; v
[1] 32.2 17.8 10.3 20.2 66.1 21.8 22.6 12.8 16.9 50.8 43.5
```

One can compute the sum of the elements of an R vector in the following two ways:

```
sage: sum(x)
[1] 47.2
sage: x.sum()
[1] 47.2
```
One can calculate the sample variance of a list of numbers:

```sage
((x-x.mean())^2/(x.length()-1)).sum()
[1] 53.853
```

```sage
x.var()
[1] 53.853
```

```sage
x.sort()
[1] 3.1 5.6 6.4 10.4 21.7
```

```sage
x.min()
[1] 3.1
```

```sage
x.max()
```

```sage
x
[1] 10.4 5.6 3.1 6.4 21.7
```

```sage
r(-17).sqrt()
[1] NaN
```

```sage
r('-17+0i').sqrt()
[1] 0+4.123106i
```

Generating an arithmetic sequence:

```sage
r('1:10')
[1] 1 2 3 4 5 6 7 8 9 10
```

Because `from` is a keyword in Python, it can’t be used as a keyword argument. Instead, `from_` can be passed, and R will recognize it as the correct thing:

```sage
r.seq(length=10, from_=-1, by=.2)
[1] -1.0 -0.8 -0.6 -0.4 -0.2 0.0 0.2 0.4 0.6 0.8
```

```sage
x = r([10.4,5.6,3.1,6.4,21.7])
```

```sage
x.rep(2)
[1] 10.4 5.6 3.1 6.4 21.7 10.4 5.6 3.1 6.4 21.7
```

```sage
x.rep(times=2)
[1] 10.4 5.6 3.1 6.4 21.7 10.4 5.6 3.1 6.4 21.7
```

```sage
x.rep(each=2)
[1] 10.4 10.4 5.6 5.6 3.1 3.1 6.4 6.4 21.7 21.7
```

Missing Values:

```sage
na = r('NA')
sage: z = r([1,2,3,na])
sage: z
[1] 1 2 3 NA
```

```sage
ind = r.is_na(z)
sage: ind
[1] FALSE FALSE FALSE TRUE
```

```sage
zero = r(0)
sage: zero / zero
[1] NaN
```

```sage
inf = r('Inf')
sage: inf-inf
[1] NaN
```

```sage
r.is_na(inf)
[1] FALSE
```

```sage
r.is_na(inf-inf)
```

(continues on next page)
Character Vectors:

```python
sage: labs = r.paste('c("X","Y")', '1:10', sep='""'); labs
[1] "X1"  "Y2"  "X3"  "Y4"  "X5"  "Y6"  "X7"  "Y8"  "X9"  "Y10"
```

Index vectors; selecting and modifying subsets of a data set:

```python
sage: na = r('NA')
sage: x = r([10.4,5.6,3.1,6.4,21.7,na]); x
[1] 10.4 5.6 3.1 6.4 21.7 NA
sage: x['!is.na(self)']
[1] 10.4 5.6 3.1 6.4 21.7

sage: x = r([10.4,5.6,3.1,6.4,21.7,na]); x
[1] 10.4 5.6 3.1 6.4 21.7 NA
sage: (x+1)[!(is.na(self)) & self>0]
[1] 11.4 4.1 0.5 22.7
```

Distributions:

```python
sage: r.options(width="60")
$width
[1] 80
sage: rr = r.dnorm(r.seq(-3,3,0.1))
sage: rr
[1] 0.004431848 0.005952532 0.007915452 0.010420935 [5] 0.013582969 0.017528300 0.022394530 0.028327038 [9] 0.035474593 0.043983596 0.053990967 0.065615815 [13] 0.078950158 0.094049077 0.110920835 0.129517596 [17] 0.149727466 0.171368592 0.194186055 0.217852177 [21] 0.241970725 0.266085250 0.289691553 0.312253933 [25] 0.333224603 0.352065327 0.381387815 0.381387815 [29] 0.391042694 0.396952547 0.408942280 0.396952547 [33] 0.391042694 0.381387815 0.368270140 0.352065327 [37] 0.333224603 0.312253933 0.289691553 0.266085250 [41] 0.241970725 0.217852177 0.194186055 0.171368592 [45] 0.149727466 0.129517596 0.110920835 0.094049077 [49] 0.078950158 0.065615815 0.053990967 0.043983596 [53] 0.035474593 0.028327038 0.022394530 0.017528300
```
Convert R Data Structures to Python/Sage:

```sage
sage: rr = r.dnorm(r.seq(-3,3,0.1))
sage: sum(rr._sage_())
9.9772125168981...
```

Or you get a dictionary to be able to access all the information:

```sage
sage: rs = r.summary(r.c(1,4,3,4,3,2,5,1))
sage: rs
Min. 1st Qu. Median Mean 3rd Qu. Max.
1.000 1.750 3.000 2.875 4.000 5.000
sage: d = rs._sage_()
sage: d['DATA']
[1, 1.75, 3, 2.875, 4, 5]
sage: d['_Names']
['Min.', '1st Qu.', 'Median', 'Mean', '3rd Qu.', 'Max. ']
sage: d['_r_class']
['summaryDefault', 'table']
```

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:
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.

EXAMPLES:

```python
sage: r = sage.interfaces.r.R()
```

```python
sage: r.summary(r.c(1,2,3,111,2,3,2,3,2,5,4))
```

```
      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.  
1.000   2.000   3.000  12.550  3.500  111.000  
```

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:

```python
sage: ap = r.available_packages()  # optional - internet
sage: len(ap) > 20  # optional - internet
True
```

call(function_name, *args, **kwds)
This is an alias for function_call().

EXAMPLES:

```python
sage: r.call('length', [1,2,3])
[1] 3
```

chdir(dir)
Changes the working directory to dir
INPUT:

• dir – the directory to change to.

EXAMPLES:

```python
sage: import tempfile
sage: tmpdir = tempfile.mkdtemp()
sage: r.chdir(tmpdir)
```

Check that `tmpdir` and `r.getwd()` refer to the same directory. We need to use `realpath()` in case $TMPDIR (by default /tmp) is a symbolic link (see trac ticket #10264).

```python
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:

```python
sage: dummy = r._tab_completion(use_disk_cache=False)  # clean doctest
sage: 'testInheritedMethods' in r.completions('tes')
True
```

`console()`

Runs the R console as a separate new R process.

EXAMPLES:

```python
sage: r.console()  # not tested
R version 2.6.1 (2007-11-26)
Copyright (C) 2007 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
...
```

`convert_r_list(l)`

Converts an R list to a Python list.

EXAMPLES:

```python
sage: s = 'c(".GlobalEnv", "package:stats", "package:graphics",
    "Autoloads", \n"package:base")

sage: r.convert_r_list(s)
```
**eval** (code, *args, **kwds)
Evaluates a command inside the R interpreter and returns the output as a string.

**EXAMPLES:**

```
sage: r.eval('1+1')
'[1] 2'
```

**function_call** (function, args=None, kwds=None)
Return the result of calling an R function, with given args and keyword args.

**OUTPUT:** RElement – an object in R

**EXAMPLES:**

```
sage: r.function_call('length', args=[[1,2,3]])
[1] 3
```

**get** (*var*)
Returns the string representation of the variable *var*.

**INPUT:**
- *var* – a string

**OUTPUT:** string

**EXAMPLES:**

```
sage: r.set('a', 2)
sage: r.get('a')
'[1] 2'
```

**help** (*command*)
Returns help string for a given command.

**INPUT:**
- *command* – a string

**OUTPUT:** HelpExpression – a subclass of string whose __repr__ method is __str__, so it prints nicely

**EXAMPLES:**

```
sage: r.help('c')
title
-----
Combine Values into a Vector or List
name
-----
c
...
```

**install_packages** (*package_name*)
Install an R package into Sage’s R installation.

**EXAMPLES:**

```
sage: r.install_packages('aaMI')  # not tested
...
```

(continues on next page)
R is free software and comes with ABSOLUTELY NO WARRANTY. You are welcome to redistribute it under certain conditions. Type 'license()' or 'licence()' for distribution details.
...
Please restart Sage in order to use 'aaMI'.

\texttt{library(library\_name)}

Load the library \texttt{library\_name} into the R interpreter.

This function raises an \texttt{ImportError} if the given library is not known.

\textbf{INPUT:}

- \texttt{library\_name} – string

\textbf{EXAMPLES:}

\begin{verbatim}
sage: r.library('grid')
sage: 'grid' in r.eval('(\packages())')
True
sage: r.library('foobar')
Traceback (most recent call last):
...
ImportError: ...
\end{verbatim}

\texttt{na()}

Returns the NA in R.

\textbf{OUTPUT:} \texttt{RElement} – an element of \texttt{R}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: r.na()
[1] NA
\end{verbatim}

\texttt{plot(*args, **kwds)}

The R plot function. Type \texttt{r.help('plot')} for much more extensive documentation about this function. See also below for a brief introduction to more plotting with R.

If one simply wants to view an R graphic, using this function is is sufficient (because it calls \texttt{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.

\textbf{EXAMPLES:}

This example saves a plot to the standard R output, usually a filename like \texttt{Rplot001.png} - from the command line, in the current directory, and in the cell directory in the notebook:

\begin{verbatim}
sage: d=r.setwd(""%s"%SAGE\_TMP)  # for doctesting only; ignore if you are trying this
sage: r.plot("1:10")  # optional -- rgraphics
null device
1
\end{verbatim}

To save to a specific file name, one should use \texttt{png()} to set the output device to that file. If this is done in the notebook, it must be done in the same cell as the plot itself:
sage: filename = tmp_filename() + '.png'
sage: r.png(filename='"%s"%filename) # Note the double quotes in single_quotes!; optional -- rgraphics
NULL
sage: x = r([1,2,3])
sage: y = r([4,5,6])
sage: r.plot(x,y) # optional -- rgraphics
null device
1
sage: import os; os.unlink(filename) # For doctesting, we remove the file;
→optional -- rgraphics

Please note that for more extensive use of R's plotting capabilities (such as the lattices package), it is advisable to either use an interactive plotting device or to use the notebook. The following examples are not tested, because they differ depending on operating system:

sage: r.X11() # not tested - opens interactive device on systems with X11
→support
sage: r.quartz() # not tested - opens interactive device on OSX
sage: r.hist("rnorm(100)") # not tested - makes a plot
sage: r.library("lattice") # not tested - loads R lattice plotting package
sage: r.histogram(x = "~ wt | cyl", data="mtcars") # not tested - makes a
→lattice plot
sage: r.dev_off() # not tested, turns off the interactive viewer

In the notebook, one can use r.png() to open the device, but would need to use the following since R lattice graphics do not automatically print away from the command line:

sage: filename = tmp_filename() + '.png' # Not needed in notebook, used for
→doctesting
sage: r.png(filename='"%s"%filename) # filename not needed in notebook, used
→for doctesting; optional -- rgraphics
NULL
sage: r.library("lattice")
sage: r("print(histogram(~wt | cyl, data=mtcars))") # plot should appear;
→optional -- rgraphics
sage: import os; os.unlink(filename) # We remove the file for doctesting, not
→needed in notebook; optional -- rgraphics

\texttt{png(*args, **kwds)}

Creates an R PNG device.

This should primarily be used to save an R graphic to a custom file. Note that when using this in the notebook, one must plot in the same cell that one creates the device. See r.plot() documentation for more information about plotting via R in Sage.

These examples won’t work on the many platforms where R still gets built without graphics support.

EXAMPLES:

sage: filename = tmp_filename() + '.png';
# optional -- rgraphics
sage: x = r([1,2,3])
sage: y = r([4,5,6])
sage: r.plot(x,y) # This saves to filename, but is not viewable from command
→line; optional -- rgraphics
null device
We want to make sure that we actually can view R graphics, which happens differently on different platforms:

```python
sage: s = r.eval('capabilities("png")')  # Should be on Linux and Solaris
sage: t = r.eval('capabilities("aqua")')  # Should be on all supported Mac versions
sage: "TRUE" in s+t  # optional -- rgraphics
True
```

### read(filename)
Read filename into the R interpreter by calling R’s source function on a read-only file connection.

**EXAMPLES:**

```python
sage: filename = tmp_filename()
sage: f = open(filename, 'w')
sage: _ = f.write('a <- 2+2

2')
sage: f.close()
sage: r.read(filename)
sage: r.get('a')
'[1] 4'
```

### require(library_name)
Load the library library_name into the R interpreter.

This function raises an ImportError if the given library is not known.

**INPUT:**

- `library_name` – string

**EXAMPLES:**

```python
sage: r.library('grid')
sage: 'grid' in r.eval('.packages()')
True
sage: r.library('foobar')
Traceback (most recent call last):
  ... ImportError: ...
```

### set(var, value)
Set the variable var in R to what the string value evaluates to in R.

**INPUT:**

- `var` – a string
- `value` – a string

**EXAMPLES:**

```python
sage: r.set('a', '2 + 3')
sage: r.get('a')
'[1] 5'
```
set_seed (seed=None)

Set the seed for R interpreter.

The seed should be an integer.

EXAMPLES:

```
sage: r = R()
sage: r.set_seed(1)
1
sage: r.sample("1:10", 5)
[1] 3 4 5 7 2
```

source (s)

Display the R source (if possible) about the function named s.

INPUT:

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

OUTPUT: string – source code

EXAMPLES:

```
sage: print(r.source("c"))
function (...) .Primitive("c")
```

dot_product (other)

Implements the notation self . other.

INPUT:

- self, other – R elements

OUTPUT: R element

EXAMPLES:

```
sage: c = r.c(1,2,3,4)
sage: c.dot_product(c.t())
[1,] 1 2 3 4
[2,] 2 4 6 8
```

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:


\[
\begin{bmatrix}
3 & 6 & 9 & 12 \\
4 & 8 & 12 & 16
\end{bmatrix}
\]

\begin{verbatim}
sage: v = r([3,-1,8])
sage: v.dot_product(v)
[1,1]
[1,1] 74
\end{verbatim}

\textbf{is\_string()} \\
Tell whether this element is a string.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: r("abc").is_string()
True
sage: r([1,2,3]).is_string()
False
\end{verbatim}

\textbf{stat\_model(x)} \\
The tilde regression operator in R.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: x = r([1,2,3,4,5])
sage: y = r([3,5,7,9,11])
sage: a = r.lm( y.tilde(x) ) # \textit{lm}( y \sim x )
sage: d = a._sage_()
sage: d['DATA']['coefficients']['DATA'][1]
2
\end{verbatim}

\textbf{tilde(x)} \\
The tilde regression operator in R.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: x = r([1,2,3,4,5])
sage: y = r([3,5,7,9,11])
sage: a = r.lm( y.tilde(x) ) # \textit{lm}( y \sim x )
sage: d = a._sage_()
sage: d['DATA']['coefficients']['DATA'][1]
2
\end{verbatim}

\textbf{class sage.interfaces.r.RFunction(parent, name, r_name=None)} \\
A Function in the R interface.

\textbf{INPUT:}

- parent – the R interface
- name – the name of the function for Python
- r_name – the name of the function in R itself (which can have dots in it)

\textbf{EXAMPLES:}

\begin{verbatim}
sage: length = r.length
sage: type(length)
\end{verbatim}
<class 'sage.interfaces.r.RFunction'>
sage: loads(dumps(length))
length

class sage.interfaces.r.RFunctionElement(obj, name)
    Bases: sage.interfaces.interface.InterfaceFunctionElement

sage.interfaces.r.is_RElement(x)
    Return True if x is an element in an R interface.

    INPUT:
    • x – object

    OUTPUT: bool

    EXAMPLES:

    sage: from sage.interfaces.r import is_RElement
    sage: is_RElement(2)
    False
    sage: is_RElement(r(2))
    True

sage.interfaces.r.r_console()
    Spawn a new R command-line session.

    EXAMPLES:

    sage: r.console()  # not tested
    R version 2.6.1 (2007-11-26)
    Copyright (C) 2007 The R Foundation for Statistical Computing
    ISBN 3-900051-07-0
    ...

sage.interfaces.r.r_version()
    Return the R version.

    EXAMPLES:

    sage: r_version()  # not tested
    ((3, 0, 1), 'R version 3.0.1 (2013-05-16)')
    sage: rint, rstr = r_version()
    sage: rint[0] >= 3
    True
    sage: rstr.startswith('R version')
    True

sage.interfaces.r.reduce_load_R()
    Used for reconstructing a copy of the R interpreter from a pickle.

    EXAMPLES:

    sage: from sage.interfaces.r import reduce_load_R
    sage: reduce_load_R()
    R Interpreter
INTERFACE TO SEVERAL RUBIK’S CUBE SOLVERS.

The first is by Michael Reid, and tries to find an optimal solution given the cube’s state, and may take a long time. See http://www.math.ucf.edu/~reid/Rubik/optimal_solver.html

The second is by Eric Dietz, and uses a standard (?) algorithm to solve the cube one level at a time. It is extremely fast, but often returns a far from optimal solution. See http://wrongway.org/?rubiksource

The third is by Dik Winter and implements Kociemba’s algorithm which finds reasonable solutions relatively quickly, and if it is kept running will eventually find the optimal solution.

AUTHOR: – Optimal was written by Michael Reid <reid@math.ucf.edu> (2004) – Cubex was written by Eric Dietz <root@wrongway.org> (2003) – Kociemba was written by Dik T. Winter <dik.winter@cwi.nl> (1993) – Initial interface by Robert Bradshaw (2007-08)

class sage.interfaces.rubik.CubexSolver

    format_cube(facets)

    solve(facets)

EXAMPLES:

    sage: from sage.interfaces.rubik import *
    sage: C = RubiksCube("R U")
    sage: CubexSolver().solve(C.facets())
    'R U'
    sage: C = RubiksCube("R U F L B D")
    sage: sol = CubexSolver().solve(C.facets()); sol
    'U' L' L' U' L U' L U L' D' L' D' L D L' U' L D' U L' U L' U L' B' U' L' U B
    → L D L' D' U' L' U L B L B' L' U L U' L' F' L' F' L' F' F' L' F' L' D' L' D D L D'
    → B L B' L B' F' L F F B' L F' B D' D' L D B' B' L' D' B U' U' L' B' D' F
    → 'F' L D F''
    sage: RubiksCube(sol) == C
    True
    sage: C = RubiksCube("R2 F'")
    sage: CubexSolver().solve(C.facets())
    "R' R' F'"
    sage: C = RubiksCube().scramble()
    sage: sol = CubexSolver().solve(C.facets())
    sage: C == RubiksCube(sol)
    True

class sage.interfaces.rubik.DikSolver

    format_cube(facets)
solve \((\text{facets, timeout}=10, \text{extra}\_\text{time}=2)\)

EXEMPLARY:

```python
sage: from sage.interfaces.rubik import *
sage: C = RubiksCube().move("R U")
sage: DikSolver().solve(C.facets())
'R U'
sage: C = RubiksCube().move("R U F L B D")
sage: DikSolver().solve(C.facets())
'R U F L B D'
sage: C = RubiksCube().move("R2 F'")
sage: DikSolver().solve(C.facets())
"R2 F'"
```

class `sage.interfaces.rubik.OptimalSolver` \((\text{verbose}=False, \text{wait}=True)\)

Interface to Michael Reid’s optimal Rubik’s Cube solver.

format_cube \((\text{facets})\)

ready ()

solve \((\text{facets})\)

The initial startup and precomputation are substantial...

TODO: Let it keep searching once it found a solution?

EXEMPLARY:

```python
sage: from sage.interfaces.rubik import *
sage: solver = DikSolver()
sage: solver = OptimalSolver()  # long time (28s on sage.math, 2012)
Initializing tables...
Done.
sage: C = RubiksCube("R U")
sage: solver.solve(C.facets())
'R U'
sage: C = RubiksCube("R U F L B D")
sage: solver.solve(C.facets())
'R U F L B D'
sage: C = RubiksCube("R2 D2")
sage: solver.solve(C.facets())
'R2 D2'
```

```python
start ()
stop ()
```

class `sage.interfaces.rubik.SingNot` \((s)\)

This class is to resolve difference between various Singmaster notation. Case is ignored, and the second and third letters may be swapped.

EXEMPLARY:

```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, sage.interfaces.expect.Expect

Expect interface to the Sage interpreter itself.

INPUT:

- **server** - (optional); if specified runs Sage on a remote machine with address. You must have ssh keys setup so you can login to the remote machine by typing “ssh remote_machine” and no password, call `_install_hints_ssh()` for hints on how to do that.

The version of Sage should be the same as on the local machine, since pickling is used to move data between the two Sage process.

EXAMPLES: We create an interface to a copy of Sage. This copy of Sage runs as an external process with its own memory space, etc.

```
sage: s = Sage()
```

Create the element 2 in our new copy of Sage, and cube it.

```
sage: a = s(2)
sage: a^3
8
```

Create a vector space of dimension 4, and compute its generators:

```
sage: V = s('QQ^4')
sage: V gens()
((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))
```

Note that V is a not a vector space, it’s a wrapper around an object (which happens to be a vector space), in another running instance of Sage.

```
sage: type(V)
<class 'sage.interfaces.sage0.SageElement'>
sage: V.parent ()
Sage
sage: g = V.0; g
(1, 0, 0, 0)
```

(continues on next page)
We can still get the actual parent by using the \texttt{name} attribute of \texttt{g}, which is the variable name of the object in the child process.

\begin{verbatim}
 sage: s('%s.parent()' % g.name())
 Vector space of dimension 4 over Rational Field
\end{verbatim}

Note that the memory space is completely different.

\begin{verbatim}
 sage: x = 10
 sage: s('x = 5')
 5
 sage: x
 10
 sage: s('x')
 5
\end{verbatim}

We can have the child interpreter itself make another child Sage process, so now three copies of Sage are running:

\begin{verbatim}
 sage: s3 = s('Sage()')
 sage: a = s3(10)
 sage: a
 10
\end{verbatim}

This \(a = 10\) is in a subprocess of a subprocess of your original Sage.

\begin{verbatim}
 sage: _ = s.eval('%s.eval("x=8")' % s3.name())
 sage: s3('x')
 8
 sage: s('x')
 5
 sage: x
 10
\end{verbatim}

The double quotes are needed because the call to \texttt{s3} first evaluates its arguments using the \texttt{s} interpreter, so the call to \texttt{s3} is passed \texttt{s('"x"')}, which is the string "\texttt{x}" in the \texttt{s} interpreter.

\begin{verbatim}
 clear(\text{\texttt{var}})
\end{verbatim}

Clear the variable named \text{\texttt{var}}.

Note that the exact format of the \texttt{NameError} for a cleared variable is slightly platform dependent, see \texttt{trac ticket #10539}.

\textbf{EXAMPLES:}

\begin{verbatim}
 sage: sage0.set('x', '2')
 sage: sage0.get('x')
 '2'
 sage: sage0.clear('x')
 sage: 'NameError' in sage0.get('x')
 True
\end{verbatim}

\begin{verbatim}
 console()
\end{verbatim}

Spawn a new Sage command-line session.
cputime 
Return cputime since this Sage subprocess was started.

```plaintext
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 
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:**
```plaintext
sage: sage0.eval('2+2')
'4'
```

get 
Get the value of the variable var.

**EXAMPLES:**
```plaintext
sage: sage0.set('x', '2')
sage: sage0.get('x')
'2'
```

new 
**EXAMPLES:**
```plaintext
sage: sage0.new(2)
2
sage: _.parent()
Sage
```

preparse 
Returns the preparsed version of the string s.

**EXAMPLES:**
```plaintext
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
```

**class sage.interfaces.sage0.SageElement**(parent, value, is_name=False, name=None)

**Bases:** sage.interfaces.expect.ExpectElement

**class sage.interfaces.sage0.SageFunction**(obj, name)

**Bases:** sage.interfaces.expect.FunctionElement

**sage.interfaces.sage0.reduce_load_Sage()**
**EXAMPLES:**

```python
sage: from sage.interfaces.sage0 import reduce_load_Sage
sage: reduce_load_Sage()
Sage
```

**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
```

**sage.interfaces.sage0.sage0_console()**
Spawn a new Sage command-line session.

**EXAMPLES:**

```python
sage: sage0_console() #not tested
------------------------------------------------------------------------------------------------------------------------
| SageMath version ..., Release Date: ...                           | Type notebook() for the GUI, and license() for information. |
|------------------------------------------------------------------------------------------------------------------------|
| ...                                                                                                                     |
```

**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:**

```python
sage: scilab.eval('2+2')
'ans =
 4.'
```

```python
sage: scilab('2+2')
4.
```

```python
sage: a = scilab(10)
```

```python
sage: a**10
1.000D+10
```

Tutorial based the MATLAB interface tutorial:

**EXAMPLES:**

```python
sage: scilab('4+10')
14.
```

```python
sage: scilab('date')
15-Feb-2010
```

```python
sage: scilab('(6+6)/3')
4.
```

```python
sage: scilab('9')^2
81.
```

```python
sage: a = scilab(10); b = scilab(20); c = scilab(30)
```

```python
sage: avg = (a+b+c)/3
```

```python
sage: my_scalar = scilab('3.1415')
```

```python
sage: my_vector1 = scilab('[1,5,7]')
```

(continues on next page)
Matrix indexing works as follows:

sage: my_vector1 = scilab('[1;5;7]') # optional - scilab
1.
5.
7.
sage: my_vector1(1) # optional - scilab
1.
sage: my_vector1(2) # optional - scilab
5.
sage: my_vector1(3) # optional - scilab
7.

Matrix indexing works as follows:

sage: my_matrix = scilab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]') # optional - scilab
8. 12. 19.
7. 3. 2.
12. 4. 23.
8. 1. 1.
sage: my_matrix(3,2) # optional - scilab
23.

Matrix indexing works as follows:

sage: my_matrix = scilab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]') # optional - scilab
8. 12. 19.
7. 3. 2.
12. 4. 23.
8. 1. 1.
sage: my_matrix(3,2) # optional - scilab
23.
4.

One can also use square brackets:

```
sage: my_matrix[3,2]  # optional - scilab
```

Setting using parenthesis cannot work (because of how the Python language works). Use square brackets or the set function:

```
sage: my_matrix = scilab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]')  # optional - scilab
sage: my_matrix.set(2,3, 1999)  # optional - scilab
sage: my_matrix  # optional - scilab
8. 12. 19.
7. 3. 1999.
12. 4. 23.
8. 1. 1.
sage: my_matrix[2,3] = -126  # optional - scilab
sage: my_matrix  # optional - scilab
8. 12. 19.
7. 3. -126.
12. 4. 23.
8. 1. 1.
```

AUTHORS:


```python
class sage.interfaces.scilab.Scilab(maxread=None, script_subdirectory=None, logfile=None, server=None, server_tmpdir=None, seed=None):
    Bases: sage.interfaces.expect.Expect

    Interface to the Scilab interpreter.

    EXAMPLES:

    sage: a = scilab('[ 1, 1, 2; 3, 5, 8; 13, 21, 33 ]')  # optional - scilab
    sage: b = scilab('[ 1; 3; 13]')  # optional - scilab
    sage: c = a * b  # optional - scilab
    sage: print(c)  # optional - scilab
    30.
    122.
    505.
```

callable()

Starts Scilab console.

```python
EXAMPLES:

sage: scilab.console()  # optional - scilab; not tested
```

eval(command, *args, **kwds)

Evaluates commands.

```python
EXAMPLES:
```
sage: scilab.eval("5") # optional - scilab
'ans = 5.

```
sage: scilab.eval("d=44") # optional - scilab 'd =
44.
```

get (var)

Get the value of the variable var.

EXAMPLES:

```
sage: scilab.eval('b=124;') # optional - scilab
'

sage: scilab.get('b') # optional - scilab
'124.'
```

sage2scilab_matrix_string (A)

Return a Scilab matrix from a Sage matrix.

INPUT: A Sage matrix with entries in the rationals or reals.

OUTPUT: A string that evaluates to an Scilab matrix.

EXAMPLES:

```
sage: M33 = MatrixSpace(QQ,3,3) # optional - scilab
sage: A = M33([1,2,3,4,5,6,7,8,0]) # optional - scilab
sage: scilab.sage2scilab_matrix_string(A) # optional - scilab
'[[1, 2, 3; 4, 5, 6; 7, 8, 0]]'
```

set (var, value)

Set the variable var to the given value.

EXAMPLES:

```
sage: scilab.set('a', 123) # optional - scilab

sage: scilab.get('a') # optional - scilab
'123.'
```

set_seed (seed=None)

Set the seed for gp interpreter.

The seed should be an integer.

EXAMPLES:

```
sage: from sage.interfaces.scilab import Scilab # optional - scilab
sage: s = Scilab() # optional - scilab
sage: s.set_seed(1) # optional - scilab
1

sage: [s.rand() for i in range(5)] # optional - scilab
[
0.6040239,
```
version()
        Returns the version of the Scilab software used.
        EXAMPLES:

        sage: scilab.version() # optional - scilab
         'scilab-...'

whos (name=None, typ=None)
        Returns information about current objects. Arguments: nam: first characters of selected names typ: name of selected Scilab variable type
        EXAMPLES:

        sage: scilab.whos("core") # optional - scilab
         'Name Type Size Bytes...'
sage: scilab.whos(typ='function') # optional - scilab
         'Name Type Size Bytes...'

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

set (i, j, x)
        Set the variable var to the given value.
        EXAMPLES:

        sage: scilab.set('c', 125) # optional - scilab
        sage: scilab.get('c') # optional - scilab
         125.'

sage.interfaces.scilab.scilab_console()
        This requires that the optional Scilab program be installed and in your PATH, but no optional Sage packages need to be installed.
        EXAMPLES:

        sage: from sage.interfaces.scilab import scilab_console # optional - scilab
        sage: scilab_console() # optional - scilab; not tested
         scilab-5.0.3
        Consortium Scilab (DIGITEO)
        Copyright (c) 1989-2008 (INRIA)
        Copyright (c) 1989-2007 (ENPC)
Startup execution:
  loading initial environment

-->2+3
ans =
  5.
-->quit

Typing quit exits the Scilab console and returns you to Sage. Scilab, like Sage, remembers its history from one
session to another.

```
sage.interfaces.scilab.scilab_version()

Return the version of Scilab installed.

EXAMPLES:
```
sage: from sage.interfaces.scilab import scilab_version # optional - scilab
sage: scilab_version() # optional - scilab
'scilab-...'
```
AUTHORS:

- David Joyner and William Stein (2005): first version
- Martin Albrecht (2006-03-05): code so singular.[tab] and x = singular(…), x.[tab] includes all singular commands.
- Martin Albrecht (2006-03-06): This patch adds the equality symbol to singular. Also fix a problem in which “” as prompt means comparison will break all further communication with Singular.
- Martin Albrecht (2006-03-13): added current_ring() and current_ring_name()
- Simon King (2010-11-23): Reduce the overhead caused by waiting for the Singular prompt by doing garbage collection differently.
- Simon King (2011-06-06): Make conversion from Singular to Sage more flexible.

42.1 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()`.

42.2 Tutorial

EXAMPLES: First we illustrate multivariate polynomial factorization:
sage: R1 = singular.ring(0, '(x,y)', 'dp')
sage: R1
polynomial ring, over a field, global ordering
  coefficients: QQ
  number of vars : 2
  block 1: ordering dp
  : names  x y
  block 2: ordering C
sage: f = singular('9x16 - 18x13y2 - 9x12y3 + 9x10y4 - 18x11y2 + 36x8y4 + 18x7y5 -
       -18x5y6 + 9x6y4 - 18x3y6 - 9x2y7 + 9y8')
sage: f
9*x^16-18*x^13*y^2-9*x^12*y^3+9*x^10*y^4-18*x^11*y^2+36*x^8*y^4+18*x^7*y^5-18*x^5*y^6+9*x^6*y^4-18*x^3*y^6-9*x^2*y^7+9*y^8
sage: f.parent()
Singular
sage: F = f.factorize(); F
[1]:
  [1]=9
  [2]=x^6-2*x^3*y^2-x^2*y^3+y^4
  [3]=-x^5+y^2
[2]:
  1,1,2
sage: F[1]
9,
  x^6-2*x^3*y^2-x^2*y^3+y^4,
  -x^5+y^2
sage: F[1][2].sage()
  x^6 - 2*x^3*y^2 - x^2*y^3 + y^4
sage: g = f.sage(); g
9*x^16 - 18*x^13*y^2 - 9*x^12*y^3 + 9*x^10*y^4 - 18*x^11*y^2 + 36*x^8*y^4 + 18*x^7*y^5 -
       -18*x^5*y^6 + 9*x^6*y^4 - 18*x^3*y^6 - 9*x^2*y^7 + 9*y^8
sage: F[1][2].sage()
  x^6 - 2*x^3*y^2 - x^2*y^3 + y^4
sage: g.parent()
Multivariate Polynomial Ring in x, y over Rational Field
We can convert \( f \) and each exponent back to Sage objects as well.

This example illustrates polynomial GCD's:

This example illustrates computation of a Groebner basis:

(continues on next page)
The following example is the same as the one in the Singular - Gap interface documentation:

```python
sage: R = singular.ring(0, '(x0,x1,x2)', 'lp')
sage: I1 = singular.ideal(['x0*x1*x2 -x0^2*x2', 'x0^2*x1*x2-x0*x1^2*x2-x0*x1*x2^2',
                         '-x0*x1-x0*x2-x1*x2'])
sage: I2 = I1.groebner()
sage: I2
x1^2*x2^2,
x0*x2^3-x1^2*x2^2+x1*x2^3,
x0*x1-x0*x2-x1*x2,
x0^2*x2-x0*x2^2-x1*x2^2
sage: I2.sage()
Ideal (x1^2*x2^2, x0*x2^3 - x1^2*x2^2 + x1*x2^3, x0*x1 - x0*x2 - x1*x2, x0^2*x2 -
               x0*x2^2 - x1*x2^2) of Multivariate Polynomial Ring in x0, x1, x2 over Rational Field
```

This example illustrates moving a polynomial from one ring to another. It also illustrates calling a method of an object with an argument.

```python
sage: R = singular.ring(0, '(x,y,z)', 'dp')
sage: f = singular('x3+y3+(x-y)*x2y2+z2')
sage: f
x^3*y^2-x^2*y^3+x^3+y^3+z^2
sage: R1 = singular.ring(0, '(x,y,z)', 'ds')
sage: f = R.fetch(f)
sage: f
z^2+x^3+y^3+x^3*y^2-x^2*y^3
```

We can calculate the Milnor number of \( f \):

```python
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.

```python
sage: H = f.jacob().jacob()
sage: H
6*x^6+6*x*y^2-2*y^3, 6*x^2*y-6*x*y^2, 0,
6*x^2*y-6*x*y^2, 6*y+2*x^3-6*x^2*y, 0,
0, 0, 2
sage: H.sage()
[6*x + 6*x*y^2 - 2*y^3 6*x^2*y - 6*x*y^2 0]
[ 6*x^2*y - 6*x*y^2 6*y + 2*x^3 - 6*x^2*y 0]
[ 0 0 2]
sage: H.det()  # This is a polynomial in Singular
72*x*y+24*x^4-72*x^3*y+72*x*y^3-24*y^4-48*x^4*y^2+64*x^3*y^3-48*x^2*y^4
sage: H.det().sage()  # This is the corresponding polynomial in Sage
72*x*y + 24*x^4 - 72*x^3*y + 72*x*y^3 - 24*y^4 - 48*x^4*y^2 + 64*x^3*y^3 - 48*x^2*y^4
```
The 1x1 and 2x2 minors:

```
sage: H.minor(1)
2,
6*y+2*x^3-6*x^2*y,
6*x^2*y-6*x*y^2,
6*x+6*x*y^2-2*y^3
sage: H.minor(2)
12*y+4*x^3-12*x^2*y,
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+12*y^4+24*x^4*y^2-32*x^3*y^3+24*x^2*y^4
```

```
sage: __=singular.eval('option(redSB)')
sage: H.minor(1).groebner()
```

### 42.3 Computing the Genus

We compute the projective genus of ideals that define curves over $\mathbb{Q}$. It is very important to load the `normal.lib` library before calling the `genus` command, or you’ll get an error message.

**EXAMPLES:**

```
sage: singular.lib('normal.lib')
sage: R = singular.ring(0,'(x,y)','dp')
sage: i2 = singular.ideal('y9 - x2*(x-1)^9 + x')
sage: i2.genus()
40
```

Note that the genus can be much smaller than the degree:

```
sage: i = singular.ideal('y9 - x2*(x-1)^9')
sage: i.genus()
0
```

### 42.4 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.

```
sage: singular.lib('poly.lib')
sage: singular.ring(32003, '(a,b,c,d,e,f)','lp')
    polynomial ring, over a field, global ordering
    // coefficients: 22/32003
    // number of vars : 6
    // block 1 : ordering lp
```
Sage Reference Manual: Interpreter Interfaces, Release 8.7

42.5 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
# 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` (maxread=None, script_subdirectory=None, logfile=None, server=None, server_tmpdir=None, seed=None)

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

Interface to the Singular interpreter.

EXAMPLES: A Groebner basis example.

```sage
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.grobner()
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:

```python
sage: singular.lib('sing.lib')
sage: singular.lib('sing.lib', reload=True)
```

clear (var)
Clear the variable named var.

EXAMPLES:

```python
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:

```python
sage: singular.get('x')
'2'
sage: a = singular(3)
sage: singular.get('x')
'`x`
```

cconsole ()
EXAMPLES:

```python
sage: singular_console() #not tested
SINGULAR / Development
A Computer Algebra System for Polynomial Computations / version 3-0-4
0<
FB Mathematik der Universitaet, D-67653 Kaiserslautern \n```

cputime (t=None)
Returns the amount of CPU time that the Singular session has used. If t is not None, then it returns the difference between the current CPU time and t.

EXAMPLES:

```python
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*x1^2*x2', '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

\textbf{current\_ring\_name} ()

Returns the Singular name of the currently active ring in Singular.

\textbf{OUTPUT}: currently active ring's name

\textbf{EXAMPLES}:

\begin{verbatim}
sage: r = PolynomialRing(GF(127),3,'xyz')
sage: r._singular_().name() == singular.current_ring_name()
True
\end{verbatim}

\textbf{eval} \((x, allow\_semicolon=True, strip=True, **kwds)\)

Send the code \(x\) to the Singular interpreter and return the output as a string.

\textbf{INPUT}:

- \(x\) - string (of code)
- \(allow\_semicolon\) - default: False; if False then raise a TypeError if the input line contains a semicolon.
- \(strip\) - ignored

\textbf{EXAMPLES}:

\begin{verbatim}
sage: singular.eval('2 > 1')
'1'
sage: singular.eval('2 + 2')
'4'
\end{verbatim}

If the verbosity level is \(> 1\) comments are also printed and not only returned.

\begin{verbatim}
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
// -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'
sage: set_verbose(1)
sage: o = singular.eval('hilb(%s)'%(s.name()))
// 1 t^0
\end{verbatim}

(continues on next page)
This is mainly useful if this method is called implicitly. Because then intermediate results, debugging outputs and printed statements are printed

```sage
o = s.hilb()
```

rather than ignored

```sage
set_verbose(0)
```

```sage
o = s.hilb()
```

**get (var)**

Get string representation of variable named var.

EXAMPLES:

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

**ideal (\*gens)**

Return the ideal generated by gens.

INPUT:

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

OUTPUT: the Singular ideal generated by the given list of gens

EXAMPLES: A Groebner basis example done in a different way.

```sage
_ = singular.eval("ring R=0,(x0,x1,x2),lp")
sage: i1 = singular.ideal([ 'x0*x1*x2 -x0^2*x2', 'x0^2*x1*x2-x0*x1^2*x2-x0*x1*x2^2', 'x0*x1-x0*x2-x1*x2'])
sage: i1
-x0^2*x2+x0*x1*x2,
```

(continues on next page)
\begin{verbatim}
x0^2*x1*x2-x0*x1^2*x2-x0*x1*x2^2,
x0*x1-x0*x2-x1*x2
\end{verbatim}

\begin{verbatim}
sage: i2 = singular.ideal('groebner%s;''i1.name())
sage: i2
x1^2*x2^2,
x0*x2^3-x1^2*x2^2+x1*x2^3,
x0*x1-x0*x2-x1*x2,
x0^2*x2-x0*x2^2-x1*x2^2
\end{verbatim}

**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:**

\begin{verbatim}
sage: singular.lib('sing.lib')
sage: singular.lib('sing.lib', reload=True)
\end{verbatim}

**list** *(x)*

Creates a list in Singular from a Sage list *x*.

**EXAMPLES:**

\begin{verbatim}
sage: singular.list([1,2])
[1]:
  1
[2]:
  2
\end{verbatim}

**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:**

\begin{verbatim}
sage: singular.lib('sing.lib')
sage: singular.lib('sing.lib', reload=True)
\end{verbatim}

**matrix** *(nrows, ncols, entries=None)*

**EXAMPLES:**

\begin{verbatim}
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
\end{verbatim}
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.

```python
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
```
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 $\mathbb{Q}(\alpha)$ 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: a = singular(3)
sage: n = a.name()
sage: del a
sage: singular.eval(n)
'3'
sage: singular.set('int', 'y', '5')
```

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')
```

(continues on next page)
**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.


**EXAMPLES:**

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

**setring** \((R)\)

Sets the current Singular ring to \(R\).

**EXAMPLES:**

```
sage: R = singular.ring(7, '(a,b)', 'ds')
sage: S = singular.ring('real', '(a,b)', 'lp')
sage: singular.current_ring()
polynomial ring, over a field, global ordering
  // coefficients: Float()
  // number of vars : 2
  // block 1 : ordering lp
  //    : names a b
  // block 2 : ordering C
sage: singular.set_ring(R)
sage: singular.current_ring()
```
polynomial ring, over a field, local ordering
// coefficients: ZZ/7
// number of vars : 2
// block 1 : ordering ds
// : names a b
// block 2 : ordering C

string(x)

Creates a Singular string from a Sage string. Note that the Sage string has to be “double-quoted”.

EXAMPLES:

```sage```
singular.string('"Sage"')
```
Sage

version()

Return the version of Singular being used.

EXAMPLES:

```sage```
singular.version()
```
"Singular ... version 4..."

```

class sage.interfaces.singular.SingularElement (parent, type, value, is_name=False)

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

EXAMPLES:

```sage```
a = singular(2)
sage: loads(dumps(a))
2

attrib(name, value=None)

Get and set attributes for self.

INPUT:

- name - string to choose the attribute
- value - boolean value or None for reading, (default:None)

VALUES: isSB - the standard basis property is set by all commands computing a standard basis like groebner, std, stdhilb etc.; used by lift, dim, degree, mult, hilb, vdim, kbase isHomog - the weight vector for homogeneous or quasihomogeneous ideals/modules isCI - complete intersection property isCM - Cohen-Macaulay property rank - set the rank of a module (see nrows) withSB - value of type ideal, resp. module, is std withHilb - value of type intvec is hilb(_,1) (see hilb) withRes - value of type list is a free resolution withDim - value of type int is the dimension (see dim) withMult - value of type int is the multiplicity (see mult)

EXAMPLES:

```sage```
P.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([z^2, y*z, y^2, x*z, x*y, x^2])
sage: Ibar = I._singular_()
sage: Ibar.attrib('isSB')
0
```
```sage```
singular.eval('vdim(%s)'%Ibar.name()) # sage7 name is random
```
(continues on next page)
// ** sage7 is no standard basis
4
sage: Ibar.attrib('isSB',1)
sage: singular.eval('vdim(%s)'%Ibar.name())
'4'

**is_string()**

Tell whether this element is a string.

**EXAMPLES:**

```
sage: singular("abc").is_string()
True
sage: singular('1').is_string()
False
```

**sage_flattened_str_list()**

**EXAMPLES:**

```
sage: R=singular.ring(0,'(x,y)','dp')
sage: RL = R.ringlist()
sage: RL.sage_flattened_str_list()
['0', 'x', 'y', 'dp', '1,1', 'C', '0', '_[1]=0']
```

**sage_global_ring()**

Return the current basering in Singular as a polynomial ring or quotient ring.

**EXAMPLES:**

```
sage: singular.eval('ring r1 = (9,x),(a,b,c,d,e,f),(M((1,2,3,0)),wp(2,3),lp)')

sage: R = singular('r1').sage_global_ring()
sage: R
Multivariate Polynomial Ring in a, b, c, d, e, f over Finite Field in x of size 3^2
sage: R.term_order()
Block term order with blocks:
(Matrix term order with matrix
[1 2]
[3 0],
Weighted degree reverse lexicographic term order with weights (2, 3),
Lexicographic term order of length 2)
```

```
sage: singular.eval('ring r2 = (0,x),(a,b,c),dp')

sage: singular('r2').sage_global_ring()
Multivariate Polynomial Ring in a, b, c over Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

```
sage: singular.eval('ring r3 = (3,z),(a,b,c),dp')

sage: 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
```
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:

```python
sage: singular.eval('ring r4 = (real,20),(a,b,c),dp')
'sing: 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”:

```python
sage: singular.eval('ring r5 = (complex,15,j),(a,b,c),dp')
'sing: 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()('j')
Traceback (most recent call last):
... NameError: name 'j' is not defined
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:

```python
sage: singular.eval('ring r7 = (0,a), (x,y), dp')
'sing: ring r7 = (0,a), (x,y), dp
sage: singular.eval('minpoly = a^2 + 1')
'sing: minpoly = a^2 + 1
sage: singular('r7').sage_global_ring()
Multivariate Polynomial Ring in x, y over Number Field in a with defining polynomial a^2 + 1
```

In our last example, the base ring is a quotient ring:

```python
sage: singular.eval('ring r6 = (9,a), (x,y,z),lp')
'sing: ring r6 = (9,a), (x,y,z),lp
sage: Q = singular('std(ideal(x^2,x+y^2+z^3))', type='qring')
sage: Q.sage_global_ring()
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)

```
\textbf{sage_matrix}(R, \text{sparse=True})
\text{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 \texttt{sage_global_ring()} is used.
• sparse - (default: True); determines whether the resulting matrix is sparse or not.

EXAMPLES:
**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:**

```python
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
```

```python
sage: P.<x,y> = PolynomialRing(QQ, 2)
sage: f = x*y**3 - 1/9 * x + 1; f
x*y^3 - 1/9*x + 1
sage: singular(f)
x*y^3-1/9*x+1
sage: P(singular(f))
x*y^3 - 1/9*x + 1
```

**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 \textit{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 \texttt{singular.ring} or set using \texttt{ring_name.set_ring}.

**EXAMPLES:**

```python
sage: R = singular.ring(0,'(x,y)')
sage: f = singular('x^3 + 3*y^11 + 5')
sage: f
x^3+3*y^11+5
sage: f.sage_polystring()
'x**3+3*y**11+5'
```

**sage\_structured\_str\_list()**

If self is a Singular list of lists of Singular elements, returns corresponding Sage list of lists of strings.

**EXAMPLES:**

```python
sage: R=singular.ring(0,'(x,y)','dp')
sage: RL=R.ringlist()
sage: RL
[1]:
  0
[2]:
  [1]:
    x
  [2]:
    y
[3]:
  [1]:
    [1]:
      dp
    [2]:
      1,1
  [2]:
    [1]:
      C
    [2]:
      0
[4]:
  _[1]=0
sage: RL.sage_structured_str_list()
['0', ['x', 'y'], [['dp', '1,\n1'], ['C', '0']], '0']
```

**set\_ring()**

Sets the current ring in Singular to be self.

**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: R.set_ring()
sage: singular.current_ring()
```

(continues on next page)
polynomial ring, over a field, local ordering
  // coefficients: ZZ/7
  // number of vars : 2
  // block 1 : ordering ds
  //    : names a b
  // block 2 : ordering C

type()

Returns the internal type of this element.

EXAMPLES:

```
sage: R = PolynomialRing(GF(2^8,'a'),2,'x')
sage: R._singular_().type()
'ring'
sage: fs = singular('x0^2','poly')
sage: fs.type()
'poly'
```

exception sage.interfaces.singular.SingularError

Bases: exceptions.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)

Within this context all Singular Groebner basis calculations are reduced automatically.

AUTHORS:

- Martin Albrecht
- Simon King

class sage.interfaces.singular.SingularGBLogPrettyPrinter (verbosity=1)

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:

```python
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:

```python
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 is x is of type SingularElement.

EXAMPLES:

```python
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:

```python
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:

```python
sage: singular_console()  #not tested
SINGULAR / Development
A Computer Algebra System for Polynomial Computations / version 3-0-4
FB Mathematik der Universitaet, D-67653 Kaiserslautern \n```

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..."```
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
sage: sx = Symbol('x')
sage: assert sin_integral(x)_sympy_() == Si(sx)
sage: assert sin_integral(x) == Si(sx)._sage_()
sage: assert sinh_integral(x)_sympy_() == Shi(sx)
sage: assert sinh_integral(x) == Shi(sx)._sage_()
sage: assert cos_integral(x)_sympy_() == Ci(sx)
sage: assert cos_integral(x) == Ci(sx)._sage_()
sage: assert cosh_integral(x)_sympy_() == Chi(sx)
sage: assert cosh_integral(x) == Chi(sx)._sage_()
sage: assert sgn(x)_sympy_() == sign(sx)
sage: assert sgn(x) == sign(sx)._sage_()
```

AUTHORS:

- Ralf Stephan (2017-10)

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:

```python
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:

```python
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
sage: from sage.interfaces.sympy import test_all
sage: test_all()
```
THE TACHYON RAY TRACER

AUTHOR:

- John E. Stone

```python
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:

```
sage: from sage.interfaces.tachyon import TachyonRT
sage: tgen = Tachyon()
sage: tgen.texture('t1')
sage: tgen.sphere((0,0,0),1,'t1')
sage: tgen.str()[30:40]
 'resolution'
sage: t = TachyonRT()
sage: import os
sage: t(tgen.str(), outfile=os.devnull)
tachyon ... Tachyon Parallel/Multiprocessor Ray Tracer...
```
CHAPTER
FORTYFIVE

INTERFACE TO TIDES

This module contains tools to write the .c files needed for TIDES [TIDES].

Tides is an integration engine based on the Taylor method. It is implemented as a c library. The user must translate its initial value problem (IVP) into a pair of .c files that will then be compiled and linked against the TIDES library. The resulting binary will produce the desired output. The tools in this module can be used to automate the generation of these files from the symbolic expression of the differential equation.

##########################################################################
# Copyright (C) 2014 Miguel Marco <mmarco@unizar.es>, Marcos Rodriguez
# <marcos@uunizar.es>
#
# Distributed under the terms of the GNU General Public License (GPL):
#
# http://www.gnu.org/licenses/
##########################################################################

AUTHORS:
- Miguel Marco (06-2014) - Implementation of tides solver
- Marcos Rodriguez (06-2014) - Implementation of tides solver
- Alberto Abad (06-2014) - tides solver
- Roberto Barrio (06-2014) - tides solver

REFERENCES:
- [ABBR2012]
- [TIDES]

sage.interfaces.tides.genfiles_mintides(integrator, driver, f, ics, initial, final, delta, torel=1e-16, tolabs=1e-16, output="")

Generate the needed files for the min_tides library.

INPUT:
- integrator – the name of the integrator file.
- driver – the name of the driver file.
- f – the function that determines the differential equation.
- ics – a list or tuple with the initial conditions.
- initial – the initial time for the integration.
- final – the final time for the integration.
- delta – the step of the output.
• tolrel – the relative tolerance.
• tolabs – the absolute tolerance.
• output – the name of the file that the compiled integrator will write to

This function creates two files, integrator and driver, that can be used later with the min_tides library [TIDES].

```
sage.interfaces.tides.genfiles_mpfr(integrator, driver, f, ics, initial, final, delta, parameters=None, parameter_values=None, dig=20, tolrel=1e-16, tolabs=1e-16, output=“”)
```

Generate the needed files for the mpfr module of the tides library.

**INPUT:**

• integrator – the name of the integrator file.
• driver – the name of the driver file.
• f – the function that determines the differential equation.
• ics – a list or tuple with the initial conditions.
• initial – the initial time for the integration.
• final – the final time for the integration.
• delta – the step of the output.
• parameters – the variables inside the function that should be treated as parameters.
• parameter_values – the values of the parameters for the particular initial value problem.
• dig – the number of digits of precision that will be used in the integration
• tolrel – the relative tolerance.
• tolabs – the absolute tolerance.
• output – the name of the file that the compiled integrator will write to

This function creates two files, integrator and driver, that can be used later with the tides library ([TIDES]).

```
sage.interfaces.tides.remove_constants(l1, l2)
```

Given two lists, remove the entries in the first that are real constants, and also the corresponding elements in the second one.

```
sage: from sage.interfaces.tides import subexpressions_list, remove_constants sage: f(a)=[1+cos(7)*a] sage: l1, l2 = subexpressions_list(f) sage: l1, l2 ([sin(7), cos(7), a*cos(7) + 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) + 1], ['mul', cos(7), a], ['add', 1, a*cos(7)])
```

```
sage.interfaces.tides.remove_repeated(l1, l2)
```

Given two lists, remove the repeated elements in l1, and the elements in l2 that are on the same position.

```
sage: from sage.interfaces.tides import subexpressions_list, remove_repeated sage: f(a)=[1 + a^2, arcsin(a)] sage: l1, l2 = subexpressions_list(f) sage: l1, l2 ([a^2, a^2 + 1, a^2, -a^2, -a^2 + 1, sqrt(-a^2 + 1), arcsin(a)], [('mul', a), ('add', 1, a^2)],
```

(continues on next page)
sage.interfaces.tides.subexpressions_list \( (f, \text{pars=}\text{None}) \)
Construct the lists with the intermediate steps on the evaluation of the function.

**INPUT:**

- \( f \) – a symbolic function of several components.
- \( \text{pars} \) – 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)
([x^2, x^2 + y, sin(x), cos(x), log(y), cos(x)/log(y)],
 [(\'mul\', x, x),
 (\'add\', y, x^2),
 (\'sin\', x),
 (\'cos\', x),
 (\'log\', y),
 (\'div\', log(y), cos(x))])
```

```python
sage: f(a)=\[\text{cos}(a), \text{arctan}(a)\]
sage: from sage.interfaces.tides import subexpressions_list
sage: subexpressions_list(f)
[[\text{sin}(a), \text{cos}(a), a^2, a^2 + 1, \text{arctan}(a)],
 [\text{\'sin\', a}, \text{\'cos\', a}, \text{\'mul\', a, a}, \text{\'add\', 1, a^2}, \text{\'atan\', a}]]
```

```python
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, 
x*y - b*z],
[(\text{'mul'}, -1, y),
(\text{'add'}, -y, x),
(\text{'mul'}, x - y, s),
(\text{'mul'}, -1, s*(x - y)),
(\text{'mul'}, -1, z),
(\text{'add'}, -z, r),
(\text{'mul'}, x, r - z),
(\text{'mul'}, -1, y),
(\text{'add'}, -y, (r - z)*x),
(\text{'mul'}, y, x),
(\text{'mul'}, z, b),
(\text{'mul'}, -1, b*z),
(\text{'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), \exp(x^2 + \sin(y))],
[(\text{'mul'}, x, x),
(\text{'sin'}, y),
(\text{'cos'}, y),
(\text{'add'}, \sin(y), x^2),
(\text{'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(verbos=False)

EXAMPLES:

```
sage: sage.interfaces.quit.expect_quitall()
sage: gp.eval('a=10')
'10'
sage: gp('a')
10
sage: sage.interfaces.quit.expect_quitall()
sage: gp('a')
a
sage: sage.interfaces.quit.expect_quitall(verbos=True)
Exiting PARI/GP interpreter with PID ... running .../bin/gp --fast --emacs --
quiet --stacksize 10000000
```

sage.interfaces.quit.invalidate_all()
Invalidate all of the expect interfaces.

This is used, e.g., by the fork-based @parallel decorator.

EXAMPLES:

```
sage: a = maxima(2); b = gp(3)
sage: a, b
(2, 3)
sage: sage.interfaces.quit.invalidate_all()
sage: a
(invalid Maxima object -- The maxima session in which this object was defined is
no longer running.)
sage: b
(invalid PARI/GP interpreter object -- The pari session in which this object was
defined is no longer running.)
```

However the maxima and gp sessions should still work out, though with their state reset:

```
sage: a = maxima(2); b = gp(3) sage: a, b (2, 3)
```

sage.interfaces.quit.is_running(pid)
Return True if and only if there is a process with id pid running.

sage.interfaces.quit.killSpawnedJobs(verbos=False)

INPUT:

- `verbose` – bool (default: False); if True, display a message each time a process is sent a kill signal

EXAMPLES:
sage: gp.eval('a=10')
'10'
sage: sage.interfaces.quit.kill_spawned_jobs(verbos=False)
sage: sage.interfaces.quit.expect_quitall()
sage: gp.eval('a=10')
'10'
sage: sage.interfaces.quit.kill_spawned_jobs(verbos=True)
Killing spawned job ...

After doing the above, we do the following to avoid confusion in other doctests:

sage: sage.interfaces.quit.expect_quitall()
sage.interfaces.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
sage.interfaces.axiom, 15
sage.interfaces.cleaner, 329
sage.interfaces.ecm, 21
sage.interfaces.expect, 9
sage.interfaces.four_ti_2, 27
sage.interfaces.fricas, 33
sage.interfaces.frobby, 45
sage.interfaces.gap, 49
sage.interfaces.gap3, 59
sage.interfaces.gfan, 67
sage.interfaces.giac, 69
sage.interfaces.gnuplot, 79
sage.interfaces.gp, 81
sage.interfaces.interface, 3
sage.interfaces.jmoldata, 91
sage.interfaces.kash, 93
sage.interfaces.latte, 101
sage.interfaces.lie, 105
sage.interfaces.lisp, 113
sage.interfaces.macauley2, 117
sage.interfaces.magma, 127
sage.interfaces.magma_free, 147
sage.interfaces.maple, 149
sage.interfaces.mathematica, 157
sage.interfaces.matlab, 165
sage.interfaces.maxima, 169
sage.interfaces.maxima_abstract, 179
sage.interfaces.maxima_lib, 197
sage.interfaces.mupad, 209
sage.interfaces.mwrank, 213
sage.interfaces.octave, 217
sage.interfaces.phc, 223
sage.interfaces.polymake, 231
sage.interfaces.povray, 243
sage.interfaces.psage, 245
sage.interfaces.qepcad, 247
sage.interfaces.qsieve, 271
sage.interfaces.quit, 331
sage.interfaces.r, 275
sage.interfaces.read_data, 333
sage.interfaces.rubik, 289
sage.interfaces.sage0, 291
sage.interfaces.sagespawn, 13
sage.interfaces.scilab, 295
sage.interfaces.singular, 301
sage.interfaces.sympy, 321
sage.interfaces.tachyon, 323
sage.interfaces.tides, 325
Symbols

__call__() (sage.interfaces.tachyon.TachyonRT method), 323
_eval_line() (sage.interfaces.polymake.Polymake method), 231
_sage__() (sage.interfaces.fricas.FriCASElement method), 38

A
A() (sage.interfaces.qepcad.qepcad_formula_factory method), 263
alexander_dual() (sage.interfaces.frobby.Frobby method), 45
all_but_finitely_many() (sage.interfaces.qepcad.qepcad_formula_factory method), 265
and_() (sage.interfaces.qepcad.qepcad_formula_factory method), 265
answer() (sage.interfaces.qepcad.Qepcad method), 255
application() (sage.interfaces.polymake.Polymake method), 233
arguments() (sage.interfaces.maxima_abstract.MaximaAbstractElementFunction method), 193
as_type() (sage.interfaces.axiom.PanAxiomElement method), 18
AsciiArtString (class in sage.interfaces.interface), 3
assign_names() (sage.interfaces.magma.MagmaElement method), 140
AssignNames() (sage.interfaces.magma.MagmaElement method), 140
associated_primes() (sage.interfaces.frobby.Frobby method), 46
assume() (sage.interfaces.qepcad.Qepcad method), 255
atomic() (sage.interfaces.qepcad.qepcad_formula_factory method), 266
Attach() (sage.interfaces.magma.Magma method), 131
attach() (sage.interfaces.magma.Magma method), 132
attach_spec() (sage.interfaces.magma.Magma method), 133
AttachSpec() (sage.interfaces.magma.Magma method), 131
attrib() (sage.interfaces.singular.SingularElement method), 313
attribute() (sage.interfaces.interface.InterfaceElement method), 5
available_packages() (sage.interfaces.r.R method), 279
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), 133
blackbox() (sage.interfaces.phc.PHC method), 223
bool() (sage.interfaces.fricas.FriCASElement method), 42
bool() (sage.interfaces.gap.GapElement_generic method), 53
bool() (sage.interfaces.gp.GpElement method), 88
bool() (sage.interfaces.interface.InterfaceElement method), 6
bool() (sage.interfaces.lisp.LispElement method), 115
bool() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 188
bool() (sage.interfaces.polymake.PolymakeElement method), 237

C
C() (sage.interfaces.qepcad.qepcad_formula_factory method), 263
call() (sage.interfaces.four_ti_2.FourTi2 method), 27
call() (sage.interfaces.interface.Interface method), 3
call() (sage.interfaces.r.R method), 279
cell() (sage.interfaces.qepcad.Qepcad method), 255
chdir() (sage.interfaces.magma.Magma method), 133
chdir() (sage.interfaces.mathematica.Mathematica method), 162
chdir() (sage.interfaces.matlab.Matlab method), 167
chdir() (sage.interfaces.maxima_abstract.MaximaAbstract method), 179
chdir() (sage.interfaces.r.R method), 279
check_expression() (in module sage.interfaces.sympy), 321
circuits() (sage.interfaces.four_ti_2.FourTi2 method), 27
classified_solution_dicts() (sage.interfaces.phc.PHC_Object method), 226
clean_output() (in module sageinterfaces.mathematica), 164
cleaner() (in module sage.interfaces.cleaner), 329
clear() (sage.interfaces.giac.Giac method), 73
clear() (sage.interfaces.interface.Interface method), 3
clear() (sage.interfaces.magma.Magma method), 134
clear() (sage.interfaces.maple.Maple method), 152
clear() (sage.interfaces.maxima.Maxima method), 176
clear() (sage.interfaces.maxima_lib.MaximaLib method), 198
clear() (sage.interfaces.octave.Octave method), 219
clear() (sage.interfaces.polymake.Polymake method), 234
clear() (sage.interfaces.sage0.Sage method), 292
clear() (sage.interfaces.singular.Singular method), 306
clear_prompts() (sage.interfaces.expect.Expect method), 9
close() (sage.interfaces.sagespawn.SagePtyProcess method), 13
cls() (sage.interfaces.macauley2.Macauley2Element method), 121
comma() (sage.interfaces.axiom.PanAxiomElement method), 18
comma() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 188
command() (sage.interfaces.expect.Expect method), 9
completions() (sage.interfaces.giac.Giac method), 73
completions() (sage.interfaces.maple.Maple method), 152
completions() (sage.interfaces.maxima_abstract.MaximaAbstract method), 180
completions() (sage.interfaces.mupad.Mupad method), 209
completions() (sage.interfaces.r.R method), 280
connected_subset() (sage.interfaces.qepcad.qepcad_formula_factory method), 266
console() (in module sage.interfaces.expect), 12
console() (sage.interfaces.axiom.Axiom method), 17
console() (sage.interfaces.fricas.FriCAS method), 36
console() (sage.interfaces.gap.Gap method), 51
console() (sage.interfaces.gap3.Gap3 method), 64
console() (sage.interfaces.giac.Giac method), 73
console() (sage.interfaces.gnuplot.Gnuplot method), 79
console() (sage.interfaces.jp.Gp method), 83
console() (sage.interfaces.interface.Interface method), 3
console() (sage.interfaces.kash.Kash method), 99
console() (sageinterfaces.lie.LiE method), 109
console() (sage.interfaces.lisp.Lisp method), 113
console() (sage.interfaces.macaulay2.Macaulay2 method), 118
console() (sage.interfaces.magma.Magma method), 134
console() (sage.interfaces.maple.Maple method), 152
console() (sage.interfaces.mathematica.Mathematica method), 162
console() (sage.interfaces.matlab.Matlab method), 167
console() (sage.interfaces.maxima_abstract.MaximaAbstract method), 180
console() (sage.interfaces.mupad.Mupad method), 209
console() (sage.interfaces.mwrank.Mwrank_class method), 213
console() (sage.interfaces.octave.Octave method), 219
console() (sage.interfaces.polymake.Polymake method), 234
console() (sage.interfaces.sage0.Sage method), 292
console() (sage.interfaces.sci.lab.SciLab method), 297
console() (sage.interfaces.singular.Singular method), 306
convert_r_list() (sage.interfaces.sage0.Sage method), 280
cputime() (sage.interfaces.gap.Gap method), 51
cputime() (sage.interfaces.gap3.Gap3 method), 64
cputime() (sage.interfaces.giac.Giac method), 73
cputime() (sage.interfaces.jp.Gp method), 83
cputime() (sage.interfaces.interface.Interface method), 3
cputime() (sage.interfaces.macaulay2.Macaulay2 method), 118
cputime() (sage.interfaces.magma.Magma method), 134
cputime() (sage.interfaces.maple.Maple method), 152
cputime() (sage.interfaces.maxima_abstract.MaximaAbstract method), 180
cputime() (sage.interfaces.mupad.Mupad method), 209
cputime() (sage.interfaces.qsieve.qsieve_nonblock method), 272
cputime() (sage.interfaces.sage0.Sage method), 293
cputime() (sage.interfaces.singular.Singular method), 306
CubexSolver (class in sage.interfaces.rubik), 289
current_ring() (sage.interfaces.singular.Singular method), 306
current_ring_name() (sage.interfaces.singular.Singular method), 307
D
data_to_list() (in module sage.interfaces.qsieve), 271
de_solve() (sage.interfaces.maxima_abstract.MaximaAbstract method), 181
de_solve_laplace() (sage.interfaces.maxima_abstract.MaximaAbstract method), 181
de_system_plot() (sage.interfaces.octave.Octave method), 220
definition() (sage.interfaces.maxima_abstract.MaximaAbstractElementFunction method), 194
demo() (sage.interfaces.maxima_abstract.MaximaAbstract method), 182
derivative() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 188
describe() (sage.interfaces.maxima_abstract.MaximaAbstract method), 182
detach() (sage.interfaces.expect.Expect method), 9
diff() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 189
DikSolver (class in sage.interfaces.rubik), 289
dimension() (sage.interfaces.frobby.Frobby method), 46
directory() (sage.interfaces.four_ti_2.FourTi2 method), 28
display2d() (sage.interfaces.maxima.MaximaElement method), 177
display2d() (sage.interfaces.maxima_lib.MaximaLibElement method), 202
done () (sage.interfaces.qsieve.qsieve_nonblock method), 272
dot() (sage.interfaces.macaulay2.Macaulay2Element method), 121
dot() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 189
dot_product() (sage.interfaces.r.RElement method), 285
dummy_integrate() (in module sage.interfaces.maxima_lib), 203

E
E() (sage.interfaces.qepcad.qepcad_formula_factory method), 264
ecl() (sage.interfaces.maxima_lib.MaximaLibElement method), 203
ECM (class in sage.interfaces.ecm), 21
eval() (sage.interfaces.expect.Expect method), 10
eval() (sage.interfaces.fricas.FriCAS method), 37
eval() (sage.interfaces.gap.Gap_generic method), 54
eval() (sage.interfaces.giac.Giac method), 74
eval () (sage.interfaces.interface.Interface method), 3
eval() (sage.interfaces.kash.Kash method), 99
eval() (sage.interfaces.lie.LiE method), 109
eval() (sage.interfaces.lisp.Lisp method), 114
eval() (sage.interfaces.macaulay2.Macaulay2 method), 118
eval() (sage.interfaces.magma.Magma method), 135
eval() (sage.interfaces.magma.MagmaElement method), 140
eval() (sage.interfaces.magma_free.MagmaFree method), 147
eval() (sage.interfaces.mathematica.Mathematica method), 162
eval() (sage.interfaces.maxima_lib.MaximaLib method), 198
eval() (sage.interfaces.mupad.Mupad method), 209
eval() (sage.interfaces.mwrank.Mwrank_class method), 213
eval() (sage.interfaces.psage.PSage method), 245
eval() (sage.interfaces.r.R method), 280
eval() (sage.interfaces.sage0.Sage method), 293
eval() (sage.interfaces.scilab.Scilab method), 297
eval() (sage.interfaces.singular.Singular method), 307
evaluate() (sage.interfaces.magma.MagmaElement method), 140
exactly_k() (sage.interfaces.qepcad.qepcad_formula_factory method), 266
evaluate() (sage.interfaces.maxima_abstract.MaximaAbstract method), 182
execute() (sage.interfaces.interface.Interface method), 4
exists() (sage.interfaces.qepcad.qepcad_formula_factory method), 267
Expect (class in sage.interfaces.expect), 9
expect() (sage.interfaces.expect.Expect method), 10
expect() (sage.interfaces.giac.Giac method), 74
expect() (sage.interfaces.maple.Maple method), 153
expect() (sage.interfaces.mupad.Mupad method), 210
expect_peek() (sage.interfaces.sagespawn.SageSpawn method), 14
expect_quitall() (in module sage.interfaces.quit), 331
expect_upto() (sage.interfaces.sagespawn.SageSpawn method), 14

342 Index
ExpectElement (class in sage.interfaces.expect), 11
ExpectFunction (class in sage.interfaces.expect), 12
export_image() (sage.interfaces.jmoldata.JmolData method), 91
extcode_dir() (in module sage.interfaces.magma), 145
external_string() (sage.interfaces.macaulay2.Macaulay2Element method), 121

F
F() (sage.interfaces.qepcad.qepcad_formula_factory method), 264
factor() (sage.interfaces.ecm.ECM method), 22
final_stats() (sage.interfaces.qepcad.Qepcad method), 255
find_factor() (sage.interfaces.ecm.ECM method), 23
flush() (sage.interfaces.magma.MagmaGBLogPrettyPrinter method), 144
flush() (sage.interfaces.singular.SingularGBLogPrettyPrinter method), 318
forall() (sage.interfaces.qepcad.qepcad_formula_factory method), 267
format_cube() (sage.interfaces.rubik.CubexSolver method), 289
format_cube() (sage.interfaces.rubik.DikSolver method), 289
format_cube() (sage.interfaces.rubik.OptimalSolver method), 290
formula() (sage.interfaces.qepcad.qepcad_formula_factory method), 268
FourTi2 (class in sage.interfaces.four_ti_2), 27
FriCAS (class in sage.interfaces.fricas), 36
fricas_console() (in module sage.interfaces.fricas), 42
FriCASElement (class in sage.interfaces.fricas), 38
FriCASExpectFunction (class in sage.interfaces.fricas), 42
FriCASElement (class in sage.interfaces.fricas), 42
Frobby (class in sage.interfaces.frobby), 45
full_typename() (sage.interfaces.polymake.PolymakeElement method), 237
function() (sage.interfaces.maxima_abstract.MaximaAbstract method), 183
function_call() (sage.interfaces.gap.Gap_generic method), 54
function_call() (sage.interfaces.interface.Interface method), 4
function_call() (sage.interfaces.lie.LiE method), 110
function_call() (sage.interfaces.lisp.Lisp method), 114
function_call() (sage.interfaces.magma.Magma method), 136
function_call() (sage.interfaces.polymake.Polymake method), 234
function_call() (sage.interfaces.r.R method), 281
FunctionElement (class in sage.interfaces.expect), 12

G
G() (sage.interfaces.qepcad.qepcad_formula_factory method), 264
Gap (class in sage.interfaces.gap), 51
Gap3 (class in sage.interfaces.gap3), 64
gap3_console() (in module sage.interfaces.gap3), 65
gap3_version() (in module sage.interfaces.gap3), 65
GAP3Element (class in sage.interfaces.gap3), 62
GAP3Record (class in sage.interfaces.gap3), 63
gap_command() (in module sage.interfaces.gap), 56
gap_console() (in module sage.interfaces.gap), 56
Gap_generic (class in sage.interfaces.gap), 53
gap_reset_workspace() (in module sage.interfaces.gap), 56
GapElement (class in sage.interfaces.gap), 53
GapElement_generic (class in sage.interfaces.gap), 53

Index 343
GapFunction (class in sage.interfaces.gap), 53
GapFunctionElement (class in sage.interfaces.gap), 53
gc_disabled (class in sage.interfaces.expect), 12
gen() (sage.interfaces.fricas.FriCASElement method), 42
gen() (sage.interfaces.interface.InterfaceElement method), 6
gen() (sage.interfaces.magma.MagmaElement method), 141
gen_names() (sage.interfaces.magma.MagmaElement method), 141
generate_docstring_dictionary() (in module sage.interfaces.singular), 318
genfiles_mintides() (in module sage.interfaces.tides), 325
genfiles_mpfr() (in module sage.interfaces.tides), 326
gens() (sage.interfaces.magma.MagmaElement method), 142
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.interface.Interface method), 4
get() (sage.interfaces.kash.Kash method), 99
get() (sage.interfaces.lie.LiE method), 110
get() (sage.interfaces.lisp.Lisp method), 114
get() (sage.interfaces.macaualay2.Macaulay2 method), 119
get() (sage.interfaces.magma.Magma method), 136
get() (sage.interfaces.maple.Maple method), 153
get() (sage.interfaces.mathematica.Mathematica method), 162
get() (sage.interfaces.matlab(Matlab method), 167
get() (sage.interfaces.maxima.Maxima method), 176
get() (sage.interfaces.maxima_lib.MaximaLib method), 199
get() (sage.interfaces.mupad.Mupad method), 210
get() (sage.interfaces.octave.Octave method), 220
get() (sage.interfaces.polymake.Polymake method), 234
get() (sage.interfaces.psage.PSage method), 245
get() (sage.interfaces.r.R method), 281
get() (sage.interfaces.sage0.Sage method), 293
get() (sage.interfaces.scilab.Scilab method), 298
get() (sage.interfaces.singular.Singular method), 308
get_boolean() (sage.interfaces.fricas.FriCAS method), 38
get_classified_solution_dicts() (in module sage.interfaces.phc), 228
get_default() (sage.interfaces.gp.Gp method), 84
get_docstring() (in module sage.interfaces.singular), 319
get_gap_memory_pool_size() (in module sage.interfaces.gap), 56
get_InputForm() (sage.interfaces.fricas.FriCAS method), 37
get_integer() (sage.interfaces.fricas.FriCAS method), 38
get_last_params() (sage.interfaces.ecm.EMC method), 24
get_magma_attribute() (sage.interfaces.magma.MagmaElement method), 142
get_member() (sage.interfaces.polymake.PolymakeElement method), 237
get_member_function() (sage.interfaces.polymake.PolymakeElement method), 238
get_nthreads() (sage.interfaces.magma.Magma method), 136
get_precision() (sage.interfaces.gp.Gp method), 84
get_real_precision() (sage.interfaces.gp.Gp method), 84
get_record_element() (sage.interfaces.gap.GapGeneric 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.phc), 228
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), 110
get_variable_list() (in module sage.interfaces.phc), 228
global() (sage.interfaces.interface.InterfaceElement method), 6
global() (sage.interfaces.interface.InterfaceFunctionElement method), 7
global() (sage.interfaces.kash.Kash method), 99
global() (sage.interfaces.lie.LiE method), 110
global() (sage.interfaces.lisp.Lisp method), 114
global() (sage.interfaces.macaulay2.Macaulay2 method), 119
global() (sage.interfaces.magma.Magma method), 137
global() (sage.interfaces.maple.Maple method), 153
global() (sage.interfaces.mathematica.Mathematica method), 163
global() (sage.interfaces.polymake.Polymake method), 235
global() (sage.interfaces.r.R method), 281
global() (sage.interfaces.tachyon.TachyonRT method), 324
glob_with_next眄al() (sage.interfaces.interface.Kash method), 100
HelpExpression (class in sage.interfaces.r), 279

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), 74
help() (sage.interfaces.gp.Gp method), 84
help() (sage.interfaces.interface.Interface method), 4
help() (sage.interfaces.interface.InterfaceFunctionElement method), 7
help() (sage.interfaces.kash.Kash method), 99
help() (sage.interfaces.lie.LiE method), 110
help() (sage.interfaces.lisp.Lisp method), 114
help() (sage.interfaces.macaulay2.Macaulay2 method), 119
help() (sage.interfaces.magma.Magma method), 137
help() (sage.interfaces.maple.Maple method), 153
help() (sage.interfaces.mathematica.Mathematica method), 163
help() (sage.interfaces.maxima_abstract.MaximaAbstract method), 183
help() (sage.interfaces.polymake.Polymake method), 235
help() (sage.interfaces.r.R method), 281
help() (sage.interfaces.tachyon.TachyonRT method), 324
help_search() (sage.interfaces.interface.Kash method), 100
HelpExpression (class in sage.interfaces.r), 279
hilbert() (sage.interfaces.four_ti_2.FourTi2 method), 28
hilbert() (sage.interfaces.frobby.Frobby method), 46

ideal() (sage.interfaces.macauly2.Macauly2 method), 119
ideal() (sage.interfaces.magma.Magma method), 137
ideal() (sage.interfaces.singular.Singular method), 308
iff() (sage.interfaces.qepcad.qepcad_formula_factory method), 268
imag() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 189
implies() (sage.interfaces.qepcad.qepcad_formula_factory method), 268
index() (sage.interfaces.qepcad.QepcadCell method), 257
infinitely_many() (sage.interfaces.qepcad.qepcad_formula_factory method), 268
install_packages() (sage.interfaces.r.R method), 281
integral() (sage.interfaces.giac.GiacElement method), 75
integral() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 189
integral() (sage.interfaces.maxima_abstract.MaximaAbstractElementFunction method), 194
integrate() (in module sage.interfaces.latte), 102
integrate() (sage.interfaces.giac.GiacElement method), 75
integrate() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 190
integrate() (sage.interfaces.maxima_abstract.MaximaAbstractElementFunction method), 194
interact() (sage.interfaces.ecm.EMC method), 24
interact() (sage.interfaces.gnuplot.Gnuplot method), 79
interact() (sage.interfaces.interface.Interface method), 4
Interface (class in sage.interfaces.interface), 3
InterfaceElement (class in sage.interfaces.interface), 5
InterfaceFunction (class in sage.interfaces.interface), 7
InterfaceFunctionElement (class in sage.interfaces.interface), 7
interrupt() (sage.interfaces.expect.Expect method), 10
interrupt() (sage.interfaces.gap.Gap_generic method), 55
intmod_gap_to_sage() (in module sage.interfaces.gap), 57
invalidate_all() (in module sage.interfaces.quit), 331
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.fricas), 43
is_GapElement() (in module sage.interfaces.gap), 58
is_GpElement() (in module sage.interfaces.gp), 88
is_InterfaceElement() (in module sage.interfaces.interface), 7
is_jvm_available() (sage.interfaces.jmoldata.JmolData method), 92
is_KashElement() (in module sage.interfaces.kash), 100
is_LiEElement() (in module sage.interfaces.lie), 111
is_LispElement() (in module sage.interfaces.lisp), 115
is_local() (sage.interfaces.expect.Expect method), 10
is_locked() (sage.interfaces.psage.PSage method), 246
is_locked() (sage.interfaces.psage.PSageElement method), 246
is_Macaulay2Element() (in module sage.interfaces.macaulay2), 124
is_MagmaElement() (in module sage.interfaces.magma), 145
is_MaximaElement() (in module sage.interfaces.maxima), 177
is_MaximaLibElement() (in module sage.interfaces.maxima_lib), 204
is_RElement() (in module sage.interfaces.r), 287
is_remote() (sage.interfaces.expect.Expect method), 10
is_running() (in module sage.interfaces.quit), 331
is_running() (sage.interfaces.expect.Expect method), 10
is_SingularElement() (in module sage.interfaces.singular), 319
is_string() (sage.interfaces.gap.GapElement_generic method), 53
is_string() (sage.interfaces.gp.GpElement method), 88
is_string() (sage.interfaces.interface.InterfaceElement method), 6
is_string() (sage.interfaces.r.RElement method), 286
is_string() (sage.interfaces.singular.SingularElement method), 314

J
JmolData (class in sage.interfaces.jmoldata), 91

K
Kash (class in sage.interfaces.kash), 99
kash_console() (in module sage.interfaces.kash), 100
kash_version() (in module sage.interfaces.kash), 100
KashDocumentation (class in sage.interfaces.kash), 100
KashElement (class in sage.interfaces.kash), 100
kill() (sage.interfaces.gp.Gp method), 85
kill() (sage.interfaces.lisp.Lisp method), 114
kill_spawned_jobs() (in module sage.interfaces.quit), 331
known_properties() (sage.interfaces.polymake.PolymakeElement method), 238

L
level() (sage.interfaces.qepcad.QepcadCell method), 258
LIB() (sage.interfaces.singular.Singular method), 306
lib() (sage.interfaces.singular.Singular method), 309
library() (sage.interfaces.r.R method), 282
LiE (class in sage.interfaces.lie), 109
lie_console() (in module sage.interfaces.lie), 111
lie_version() (in module sage.interfaces.lie), 111
LiEElement (class in sage.interfaces.lie), 111
LiEFunction (class in sage.interfaces.lie), 111
LiEFunctionElement (class in sage.interfaces.lie), 111
Lisp (class in sage.interfaces.lisp), 113
lisp() (sage.interfaces.maxima.Maxima method), 176
lisp() (sage.interfaces.maxima_lib.MaximaLib method), 199
lisp_console() (in module sage.interfaces.lisp), 115
LispElement (class in sage.interfaces.lisp), 115
LispFunction (class in sage.interfaces.lisp), 115
LispFunctionElement (class in sage.interfaces.lisp), 115
list() (sage.interfaces.qsieve.qsieve_nonblock method), 272
list() (sage.interfaces.singular.Singular method), 309
list_attributes() (sage.interfaces.magma.MagmaElement method), 142
load() (sage.interfaces.magma.Magma method), 138
load() (sage.interfaces.maple.Maple method), 153
load() (sage.interfaces.singular.Singular method), 309
load_package() (sage.interfaces.gap.Gap_generic method), 55
log() (sage.interfaces.qsieve.qsieve_nonblock method), 272

M

Macaulay2 (class in sage.interfaces.macaulay2), 118
macaulay2_console() (in module sage.interfaces.macaulay2), 124
Macaulay2Element (class in sage.interfaces.macaulay2), 121
Macaulay2Function (class in sage.interfaces.macaulay2), 124
Magma (class in sage.interfaces.magma), 130
magma_console() (in module sage.interfaces.magma), 145
magma_free_eval() (in module sage.interfaces.magma_free), 147
magma_gb_standard_options() (in module sage.interfaces.magma), 145
MagmaElement (class in sage.interfaces.magma), 140
MagmaExpr (class in sage.interfaces.magma_free), 147
MagmaFree (class in sage.interfaces.magma_free), 147
MagmaFunction (class in sage.interfaces.magma), 144
MagmaFunctionElement (class in sage.interfaces.magma), 144
MagmaGBDefaultContext (class in sage.interfaces.magma), 144
MagmaGBLogPrettyPrinter (class in sage.interfaces.magma), 144
make_cells() (sage.interfaces.qepcad.Qepcad method), 256
Maple (class in sage.interfaces.maple), 151
maple_console() (in module sage.interfaces.maple), 155
MapleElement (class in sage.interfaces.maple), 154
MapleFunction (class in sage.interfaces.maple), 155
MapleFunctionElement (class in sage.interfaces.maple), 155
Mathematica (class in sage.interfaces.mathematica), 162
mathematica_console() (in module sage.interfaces.mathematica), 164
MathematicaElement (class in sage.interfaces.mathematica), 163
MathematicaFunction (class in sage.interfaces.mathematica), 164
MathematicaFunctionElement (class in sage.interfaces.mathematica), 164
Matlab (class in sage.interfaces.matlab), 167
matlab_console() (in module sage.interfaces.matlab), 168
matlab_version() (in module sage.interfaces.matlab), 168
MatlabElement (class in sage.interfaces.matlab), 168
matrix() (sage.interfaces.singular.Singular method), 309
max_at_to_sage() (in module sage.interfaces.maxima_lib), 204
max_harmonic_to_sage() (in module sage.interfaces.maxima_lib), 204
max_to_sr() (in module sage.interfaces.maxima_lib), 205
max_to_string() (in module sage.interfaces.maxima_lib), 205
Maxima (class in sage.interfaces.maxima), 175
maxima_console() (in module sage.interfaces.maxima_abstract), 194
maxima_version() (in module sage.interfaces.maxima_abstract), 195
MaximaAbstract (class in sage.interfaces.maxima_abstract), 179
MaximaAbstractElement (class in sage.interfaces.maxima_abstract), 187
MaximaAbstractElementFunction (class in sage.interfaces.maxima_abstract), 193
MaximaElement (class in sage.interfaces.maxima), 177
MaximaElementFunction (class in sage.interfaces.maxima), 177
MaximaLib (class in sage.interfaces.maxima_lib), 198
MaximaLibElement (class in sage.interfaces.maxima_lib), 202
MaximaLibElementFunction (class in sage.interfaces.maxima_lib), 203
mdiff_to_sage() (in module sage.interfaces.maxima_lib), 205
methods() (sage.interfaces.magma.MagmaElement method), 143
minimize() (sage.interfaces.four_ti_2.FourTi2 method), 29
mixed_volume() (sage.interfaces.phc.PHC method), 224
mlist_to_sage() (in module sage.interfaces.maxima_lib), 205
mqapply_to_sage() (in module sage.interfaces.maxima_lib), 206
mrat_to_sage() (in module sage.interfaces.maxima_lib), 206
Mupad (class in sage.interfaces.mupad), 209
mupad_console() (in module sage.interfaces.mupad), 210
MupadElement (class in sage.interfaces.mupad), 210
MupadFunction (class in sage.interfaces.mupad), 210
MupadFunctionElement (class in sage.interfaces.mupad), 210
Mwrank() (in module sage.interfaces.mwrank), 213
Mwrank_class (class in sage.interfaces.mwrank), 213
mwrank_console() (in module sage.interfaces.mwrank), 214

N
N() (sage.interfaces.mathematica.MathematicaElement method), 163
n() (sage.interfaces.mathematica.MathematicaElement method), 163
n() (sage.interfaces.qsieve.qsieve_nonblock method), 272
na() (sage.interfaces.r.R method), 282
name() (sage.interfaces.interface.Interface method), 4
name() (sage.interfaces.interface.InterfaceElement method), 6
new() (sage.interfaces.interface.Interface method), 4
new() (sage.interfaces.sage0.Sage method), 293
new_from() (sage.interfaces.macaulay2.Macaulay2 method), 119
new_object() (sage.interfaces.polymake.Polymake method), 235
new_with_bits_prec() (sage.interfaces.gp.Gp method), 85
nintegral() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 191
not_() (sage.interfaces.qepcad.qepcad_formula_factory method), 269
number_of_children() (sage.interfaces.qepcad.QepcadCell method), 258
numer() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 192

O
objgens() (sage.interfaces.magma.Magma method), 138
Octave (class in sage.interfaces.octave), 219
octave_console() (in module sage.interfaces.octave), 222
OctaveElement (class in sage.interfaces.octave), 222
one_curve() (sage.interfaces.ecm.EMC method), 24
operations() (sage.interfaces.gap3.GAP3Record method), 63
OptimalSolver (class in sage.interfaces.rubik), 290
option() (sage.interfaces.singular.Singular method), 310
or_() (sage.interfaces.qepcad.qepcad_formula_factory method), 269

P
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), 206
partial_fraction_decomposition() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 192

Index 349
path() (sage.interfaces.expect.Expect method), 10
path_track() (sage.interfaces.phec.PHC method), 224
phase() (sage.interfaces.qepcad.Qepcad method), 256
PHC (class in sage.interfaces.phec), 223
PHC_Object (class in sage.interfaces.phec), 226
pid() (sage.interfaces.expect.Expect method), 10
pid() (sage.interfaces.qsieve.qsieve_nonblock method), 272
plot() (sage.interfaces.gnuplot.Gnuplot method), 79
plot() (sage.interfaces.r.R method), 282
plot() (sage.interfaces.qsieve.qsieve_nonblock method), 272
plot2d() (sage.interfaces.maxima_abstract.MaximaAbstract method), 184
plot2d_parametric() (sage.interfaces.maxima_abstract.MaximaAbstract method), 184
plot3d() (sage.interfaces.gnuplot.Gnuplot method), 79
plot3d() (sage.interfaces.maxima_abstract.MaximaAbstract method), 184
plot3d_parametric() (sage.interfaces.maxima_abstract.MaximaAbstract method), 184
plot3d_parametric() (sage.interfaces.maxima_abstract.MaximaAbstract method), 185
plot_list() (sage.interfaces.maxima_abstract.MaximaAbstract method), 185
plot_multilist() (sage.interfaces.maxima_abstract.MaximaAbstract method), 186
plot_paths_2d() (sage.interfaces.phec.PHC method), 225
png() (sage.interfaces.r.R method), 283
Polymake (class in sage.interfaces.polymake), 231
polymake_console() (in module sage.interfaces.polymake), 241
PolymakeElement (class in sage.interfaces.polymake), 237
PolymakeError, 240
PolymakeFunctionElement (class in sage.interfaces.polymake), 240
POVRay (class in sage.interfaces.povray), 243
ppi() (sage.interfaces.four_ti_2.FourTi2 method), 29
preparse() (sage.interfaces.sage0.Sage method), 293
PSage (class in sage.interfaces.psage), 245
PSageElement (class in sage.interfaces.psage), 246
pyobject_to_max() (in module sage.interfaces.maxima_lib), 207
Q
Qepcad (class in sage.interfaces.qepcad), 255
qepcad() (in module sage.interfaces.qepcad), 260
qepcad_banner() (in module sage.interfaces.qepcad), 262
qepcad_console() (in module sage.interfaces.qepcad), 263
Qepcad_expect (class in sage.interfaces.qepcad), 260
qepcad_formula_factory (class in sage.interfaces.qepcad), 263
qepcad_version() (in module sage.interfaces.qepcad), 270
QepcadCell (class in sage.interfaces.qepcad), 257
QepcadFunction (class in sage.interfaces.qepcad), 260
qformula (class in sage.interfaces.qepcad), 270
qsieve() (in module sage.interfaces.qsieve), 271
qsieve_block() (in module sage.interfaces.qsieve), 272
qsieve_nonblock (class in sage.interfaces.qsieve), 272
qsolve() (sage.interfaces.four_ti_2.FourTi2 method), 29
qualified_typename() (sage.interfaces.polymake.PolymakeElement method), 239
quantifier() (sage.interfaces.qepcad.qepcad_formula_factory method), 269
quit() (sage.interfaces.expect.Expect method), 11
quit() (sage.interfaces.octave.Octave method), 220
Index 351
RFunctionElement (class in sage.interfaces.r), 287
ring() (sage.interfaces.macaulay2.Macaulay2 method), 120
ring() (sage.interfaces.singular.Singular method), 310

S
Sage (class in sage.interfaces.sage0), 291
sage() (sage.interfaces.interface.InterfaceElement method), 7
sage.interfaces.axiom (module), 15
sage.interfaces.cleaner (module), 329
sage.interfaces.ecm (module), 21
sage.interfaces.expect (module), 9
sage.interfaces.fourTi2 (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.gnuplot (module), 79
sage.interfaces.gp (module), 81
sage.interfaces.interface (module), 3
sage.interfaces.jmoldata (module), 91
sage.interfaces.kash (module), 93
sage.interfaces.latte (module), 101
sage.interfaces.lie (module), 105
sage.interfaces.lisp (module), 113
sage.interfaces.macaulay2 (module), 117
sage.interfaces.magma (module), 127
sage.interfaces.magma_free (module), 147
sage.interfaces.maple (module), 149
sage.interfaces.mathematica (module), 157
sage.interfaces.matlab (module), 165
sage.interfaces.maxima (module), 169
sage.interfaces.maxima_abstract (module), 179
sage.interfaces.maxima_lib (module), 197
sage.interfaces.mupad (module), 209
sage.interfaces.mwrank (module), 213
sage.interfaces.octave (module), 217
sage.interfaces.phc (module), 223
sage.interfaces.polymake (module), 231
sage.interfaces.povray (module), 243
sage.interfaces.p sage (module), 245
sage.interfaces.qepcad (module), 247
sage.interfaces.qsieve (module), 271
sage.interfaces.quit (module), 331
sage.interfaces.r (module), 275
sage.interfaces.read_data (module), 333
sage.interfaces.rubik (module), 289
sage.interfaces.sage0 (module), 291
sage.interfaces.sagespawn (module), 13
set() (sage.interfaces.octave.Octave method), 221
set() (sage.interfaces.polymake.Polymake method), 236
set() (sage.interfaces.psage.PSage method), 246
set() (sage.interfaces.r.R method), 284
set() (sage.interfaces.sage0.Sage method), 293
set() (sage.interfaces.scilab.Scilab method), 298
set() (sage.interfaces.scilab.ScilabElement method), 299
set() (sage.interfaces.singular.Singular method), 311
set_default() (sage.interfaces.gp.Gp method), 86
set_gap_memory_pool_size() (in module sage.interfaces.gap), 58
set_magma_attribute() (sage.interfaces.magma.MagmaElement method), 144
set_nthreads() (sage.interfaces.magma.Magma method), 139
set_precision() (sage.interfaces.gp.Gp method), 86
set_real_precision() (sage.interfaces.gp.Gp method), 86
set_ring() (sage.interfaces.singular.Singular method), 312
set_ring() (sage.interfaces.singular.SingularElement method), 317
set_seed() (sage.interfaces.gap.Gap method), 52
set_seed() (sage.interfaces.gp.Gp method), 87
set_seed() (sage.interfaces.interface.Interface method), 5
set_seed() (sage.interfaces.magma.Magma method), 139
set_seed() (sage.interfaces.maxima.Maxima method), 176
set_seed() (sage.interfaces.octave.Octave method), 221
set_seed() (sage.interfaces.r.R method), 284
set_seed() (sage.interfaces.scilab.Scilab method), 298
set_seed() (sage.interfaces.singular.Singular method), 312
set_series_precision() (sage.interfaces.gp.Gp method), 87
set_server_and_command() (sage.interfaces.expect.Expect method), 11
set_truth() (sage.interfaces.qepcad.QepcadCell method), 259
set_truth_value() (sage.interfaces.qepcad.Qepcad method), 256
set_verbose() (sage.interfaces.magma.Magma method), 139
SetNthreads() (sage.interfaces.magma.Magma method), 132
setring() (sage.interfaces.singular.Singular method), 312
SetVerbose() (sage.interfaces.magma.Magma method), 132
sharp() (sage.interfaces.macaulay2.Macaulay2Element method), 122
show() (sage.interfaces.mathematica.MathematicaElement method), 164
signs() (sage.interfaces.qepcad.QepcadCell method), 259
SingNot (class in sage.interfaces.rubik), 290
Singular (class in sage.interfaces.singular), 305
singular_console() (in module sage.interfaces.singular), 319
singular_gb_standard_options() (in module sage.interfaces.singular), 319
singular_version() (in module sage.interfaces.singular), 319
SingularElement (class in sage.interfaces.singular), 313
SingularError, 318
SingularFunction (class in sage.interfaces.singular), 318
SingularFunctionElement (class in sage.interfaces.singular), 318
SingularGBDefaultContext (class in sage.interfaces.singular), 318
SingularGBLogPrettyPrinter (class in sage.interfaces.singular), 318
solution_dicts() (sage.interfaces.phc.PHC_Object method), 227
solution_extension() (sage.interfaces.qepcad.Qepcad method), 257
solutions() (sage.interfaces.phc.PHC_Object method), 227
solve() (sage.interfaces.rubik.CubexSolver method), 289
solve() (sage.interfaces.rubik.DikSolver method), 289
solve() (sage.interfaces.rubik.OptimalSolver method), 290
solve_linear() (sage.interfaces.maxima_abstract.MaximaAbstract method), 186
solve_linear_system() (sage.interfaces.octave.Octave method), 221
source() (sage.interfaces.maple.Maple method), 154
source() (sage.interfaces.r.R method), 285
sr_integral() (sage.interfaces.maxima_lib.MaximaLib method), 200
sr_limit() (sage.interfaces.maxima_lib.MaximaLib method), 201
sr_prod() (sage.interfaces.maxima_lib.MaximaLib method), 201
sr_sum() (sage.interfaces.maxima_lib.MaximaLib method), 201
sr_tlimit() (sage.interfaces.maxima_lib.MaximaLib method), 202
sr_to_max() (in module sage.interfaces.maxima_lib), 207
start() (sage.interfaces.rubik.OptimalSolver method), 290
start_cleaner() (in module sage.interfaces.cleaner), 329
start_from() (sage.interfaces.phc.PHC method), 225
stat_model() (sage.interfaces.r.RElement method), 286
stdout_to_string() (in module sage.interfaces.maxima_lib), 208
StdOutContext (class in sage.interfaces.expect), 12
stop() (sage.interfaces.rubik.OptimalSolver method), 290
str() (sage.interfaces.gap.GapElement method), 53
str() (sage.interfaces.mathematica.MathematicaElement method), 164
str() (sage.interfaces.maxima_abstract.MaximaAbstractElement method), 192
string() (sage.interfaces.singular.Singular method), 313
strip_answer() (sage.interfaces.matlab.Matlab method), 168
structure_sheaf() (sage.interfaces.macaulay2.Macaulay2Element method), 122
sub() (sage.interfaces.magma.MagmaElement method), 144
subexpressions_list() (in module sage.interfaces.tides), 327
subs() (sage.interfaces.macaulay2.Macaulay2Element method), 122
substitue() (sage.interfaces.macaulay2.Macaulay2Element method), 123
sum() (sage.interfaces.giac.GiacElement method), 76
sympy_init() (in module sage.interfaces.sympy), 321
sympy_set_to_list() (in module sage.interfaces.sympy), 321

T

TachyonRT (class in sage.interfaces.tachyon), 323
temp_project() (sage.interfaces.four_ti_2.FourTi2 method), 30
terminate_async() (sage.interfaces.sagespawn.SagePtyProcess method), 13
test_all() (in module sage.interfaces.sympy), 322
tilde() (sage.interfaces.r.RElement method), 322
time() (sage.interfaces.ecm.ECM method), 25
time() (sage.interfaces.qsieve.qsieve_nonblock method), 273
to_complex() (in module sage.interfaces.octave), 222
to_latte_polynomial() (in module sage.interfaces.latte), 103
to_poly_solve() (sage.interfaces.maxima_lib.MaximaLibElement method), 203
to_sage() (sage.interfaces.macaulay2.Macaulay2Element method), 123
type() (sage.interfaces.axiom.PanAxiomElement method), 18
type() (sage.interfaces.lie.LiEElement method), 111

Index
type() (sage.interfaces.singular.SingularElement method), 318
typedef() (sage.interfaces.polymake.PolymakeElement method), 240
typeof() (sage.interfaces.polymake.PolymakeElement method), 240

U
unapply() (sage.interfaces.giac.GiacElement method), 76
unbind() (sage.interfaces.gap.Gap_generic method), 55
underscore() (sage.interfaces.macaulay2.Macaulay2Element method), 124
unit_quadratic_integer() (sage.interfaces.maxima_abstract.MaximaAbstract method), 187
unparsed_input_form() (sage.interfaces.axiom.PanAxiomElement method), 18
usage() (sage.interfaces.povray.POVRay method), 243
usage() (sage.interfaces.tachyon.TachyonRT method), 324
use() (sage.interfaces.macaulay2.Macaulay2 method), 120
user_dir() (sage.interfaces.expect.Expect method), 11

V
validate_mwrank_input() (in module sage.interfaces.mwrank), 214
variable_list() (sage.interfaces.phc.PHC_Object method), 227
version() (sage.interfaces.gap.Gap_generic method), 56
version() (sage.interfaces.giac.Giac method), 75
version() (sage.interfaces.gp.Gp method), 87
version() (sage.interfaces.kash.Kash method), 100
version() (sage.interfaces.lie.LiE method), 111
version() (sage.interfaces.lisp.Lisp method), 115
version() (sage.interfaces.macaulay2.Macaulay2 method), 121
version() (sage.interfaces.magma.Magma method), 139
version() (sage.interfaces.matlab.Matlab method), 168
version() (sage.interfaces.maxima_abstract.MaximaAbstract method), 187
version() (sage.interfaces.octave.Octave method), 221
version() (sage.interfaces.polymake.Polymake method), 236
version() (sage.interfaces.r.R method), 285
version() (sage.interfaces.sage0.Sage method), 294
version() (sage.interfaces.scilab.Scilab method), 299
version() (sage.interfaces.singular.Singular method), 313

W
whos() (sage.interfaces.matlab.Matlab method), 168
whos() (sage.interfaces.scilab.Scilab method), 299
with_package() (sage.interfaces.maple.Maple method), 154
write() (sage.interfaces.magma.MagmaGBLogPrettyPrinter method), 144
write() (sage.interfaces.singular.SingularGBLogPrettyPrinter method), 318
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), 30

X
X() (sage.interfaces.qepcad.qepcad_formula_factory method), 265

Z
zsolve() (sage.interfaces.four_ti_2.FourTi2 method), 31