Sage Reference Manual: The Sage Command Line

Release 6.3

The Sage Development Team

August 11, 2014

CONTENTS

CHAPTER

ONE

INVOKING SAGE

To run Sage, you basically just need to type `sage` from the command-line prompt to start the Sage interpreter. See the Sage Installation Guide for information about making sure your `$PATH` is set correctly, etc.

1.1 Command-line options for Sage

Running Sage, the most common options

- `file.[sage|py|spyx]` – run the given .sage, .py or .spyx files (as in `sage my_file.sage`)
- `-h`, `-?`, `--help` – print a short help message
- `-v`, `--version` – print the Sage version
- `--advanced` – print (essentially this) list of Sage options
- `-c cmd` – evaluate `cmd` as sage code. For example, `sage -c 'print factor(35)'` will print "5 * 7".

Running Sage, other options

- `--preparse file.sage` – preparse `file.sage`, a file of Sage code, and produce the corresponding Python file `file.sage.py`. See the Sage tutorial for more about preparsing and the differences between Sage and Python.
- `-q` – quiet; start with no banner
- `--grep [options] <string>` – grep through all the Sage library code for `string`. Any options will get passed to the "grep" command; for example, `sage --grep -i epstein` will search for `epstein`, and the `-i` flag tells grep to ignore case when searching. Note that while running Sage, you can also use the function `search_src` to accomplish the same thing.
- `--grepdoc [options] <string>` – grep through all the Sage documentation for `string`. Note that while running Sage, you can also use the function `search_doc` to accomplish the same thing.
- `--min [...]` – do not populate global namespace (must be first option)
- `-gthread`, `-qthread`, `-q4thread`, `-wthread`, `-pylab` – pass the option through to IPython
- `--nodotsage` – run Sage without using the user's `.sage` directory: create and use a temporary `.sage` directory instead. Warning: notebooks are stored in the `.sage` directory, so any notebooks created while running with `--nodotsage` will be temporary also.

Running the notebook

- `-n`, `--notebook` – start the Sage notebook, passing all remaining arguments to the 'notebook' command in Sage
- `-bn [...]`, `--build-and-notebook [...]` – build the Sage library (as by running `sage -b`) then start the Sage notebook
- `--inotebook [...]` – start the *insecure* Sage notebook

Running external programs and utilities

- `--cython [...]` – run Cython with the given arguments
- `--ecl [...]`, `--lisp [...]` – run Sage's copy of ECL (Embeddable Common Lisp) with the given arguments
- `--gap [...]` – run Sage's Gap with the given arguments
- `--git [...]` – run Sage's Git with the given arguments
- `--gp [...]` – run Sage's PARI/GP calculator with the given arguments
- `--ipython [...]` – run Sage's IPython using the default environment (not Sage), passing additional options to IPython
- `--kash [...]` – run Sage's Kash with the given arguments
- `--M2 [...]` – run Sage's Macaulay2 with the given arguments
- `--maxima [...]` – run Sage's Maxima with the given arguments
- `--mwrank [...]` – run Sage's mwrank with the given arguments
- `--python [...]` – run the Python interpreter
- `-R [...]` – run Sage's R with the given arguments
- `--scons [...]` – run Sage's scons
- `--singular [...]` – run Sage's singular with the given arguments
- `--twistd [...]` – run Twisted server
- `--sh [...]` – run a shell with Sage environment variables set
- `--gdb` – run Sage under the control of gdb
- `--gdb-ipython` – run Sage's IPython under the control of gdb
- `--cleaner` – run the Sage cleaner. This cleans up after Sage, removing temporary directories and spawned processes. (This gets run by Sage automatically, so it is usually not necessary to run it separately.)

Installing packages and upgrading

- `-i [options] [packages]` – install the given Sage packages (unless they are already installed); if no packages are given, print a list of all installed packages. Options:
  - `-c` – run the packages' test suites, overriding the settings of `SAGE_CHECK` and `SAGE_CHECK_PACKAGES`.
  - `-f` – force build: install the packages even if they are already installed.

  - `-s` – do not delete the `spkg/build` directories after a successful build – useful for debugging.
- `-f [options] [packages]` – shortcut for `-i -f`: force build of the given Sage packages.
- `--info [packages]` – display the `SPKG.txt` file of the given Sage packages.
- `--standard` – list all standard packages that can be installed
- `--optional` – list all optional packages that can be installed
- `--experimental` – list all experimental packages that can be installed
- `--upgrade [url]` – download, build and install standard packages from given url. If url not given, automatically selects a suitable mirror. If url='ask', it lets you select the mirror.

Building and testing the Sage library

- `--root` – print the Sage root directory
- `--branch` – print the current Sage branch
- `--clone [new branch]` – clone a new branch of the Sage library from the current branch
- `-b [branch]` – build Sage library – do this if you have modified any source code files in `$SAGE_ROOT/devel/sage/`. If `branch` is given, switch to the branch in `$SAGE_ROOT/devel/sage-branch` and build it.
- `-ba [branch]` – same as `-b`, but rebuild *all* Cython code. This could take a while, so you will be asked if you want to proceed.
- `-ba-force [branch]` – same as `-ba`, but don't query before rebuilding
- `--br [branch]` – switch to, build, and run Sage with the given branch
- `-t [options] <files|dir>` – test examples in .py, .pyx, .sage or .tex files. Options:
  - `--long` – include lines with the phrase 'long time'
  - `--verbose` – print debugging output during the test
  - `--optional` – also test all examples labeled `# optional`
  - `--only-optional[=tags]` – if no `tags` are specified, only run blocks of tests containing a line labeled `# optional`. If a comma separated list of tags is specified, only run blocks containing a line labeled `# optional tag` for any of the tags given and in these blocks only run the lines which are unlabeled or labeled `#optional` or labeled `#optional tag` for any of the tags given.
  - `--randorder[=seed]` – randomize order of tests
- `-tnew [...]` – like `-t` above, but only tests files modified since last commit
- `-tp <N> [...]` – like `-t` above, but tests in parallel using `N` threads with 0 interpreted as `minimum(8, cpu_count())`
- `--testall [options]` – test all source files, docs, and examples; options are the same as for `-t`.
- `-bt [...]` – build and test, options like `-t` above
- `-btp <N> [...]` – build and test in parallel, options like `-tp` above
- `-btnew [...]` – build and test modified files, options like `-tnew`
- `--fixdoctests file.py [output_file] [--long]` – writes a new version of `file.py` to `output_file` (default: `file.py.out`) that will pass the doctests. With the optional `--long` argument the long time tests are also checked. A patch for the new file is printed to stdout.

- `--startuptime [module]` – display how long each component of Sage takes to start up. Optionally specify a module (e.g., "sage.rings.qqbar") to get more details about that particular module.
- `--coverage <files>` – give information about doctest coverage of files
- `--coverageall` – give summary info about doctest coverage of all files in the Sage library

Documentation

- `--docbuild [options] document (format | command)` – build or return information about the Sage documentation.
  - `document` – name of the document to build
  - `format` – document output format
  - `command` – document-specific command

  A `document` and either a `format` or a `command` are required, unless a list of one or more of these is requested.

  Options:
  - `help`, `-h`, `--help` – print a help message
  - `-H`, `--help-all` – print an extended help message, including the output from the options `-h`, `-D`, `-F`, `-C all`, and a short list of examples.
  - `-D`, `--documents` – list all available documents
  - `-F`, `--formats` – list all output formats
  - `-C DOC`, `--commands=DOC` – list all commands for document `DOC`; use `-C all` to list all
  - `-i`, `--inherited` – include inherited members in reference manual; may be slow, may fail for PDF output
  - `-u`, `--underscore` – include variables prefixed with _ in reference manual; may be slow, may fail for PDF output
  - `-j`, `--jsmath` – render math using jsMath; formats: `html`, `json`, `pickle`, `web`
  - `--no-pdf-links` – do not include PDF links in document `website`; formats: `html`, `json`, `pickle`, `web`
  - `--check-nested` – check picklability of nested classes in document `reference`
  - `-N`, `--no-colors` – do not color output; does not affect children
  - `-q`, `--quiet` – work quietly; same as `--verbose=0`
  - `-v LEVEL`, `--verbose=LEVEL` – report progress at level 0 (quiet), 1 (normal), 2 (info), or 3 (debug); does not affect children

  Advanced – use these options with care:
  - `-S OPTS`, `--sphinx-opts=OPTS` – pass comma-separated `OPTS` to sphinx-build
  - `-U`, `--update-mtimes` – before building reference manual, update modification times for auto-generated ReST files

Making Sage packages or distributions

- `--pkg dir` – create the Sage package `dir.spkg` from the directory `dir`
- `--pkg_nc dir` – as `--pkg`, but do not compress the package
- `--merge` – run Sage's automatic merge and test script
- `--bdist VER` – build a binary distribution of Sage, with version `VER`
- `--sdist` – build a source distribution of Sage
- `--crap sage-ver.tar` – detect suspicious garbage in the Sage source tarball

Valgrind memory debugging

- `--cachegrind` – run Sage using Valgrind's cachegrind tool
- `--callgrind` – run Sage using Valgrind's callgrind tool
- `--massif` – run Sage using Valgrind's massif tool
- `--memcheck` – run Sage using Valgrind's memcheck tool
- `--omega` – run Sage using Valgrind's omega tool
- `--valgrind` – this is an alias for `--memcheck`

CHAPTER

TWO

SAGE STARTUP SCRIPTS

There are two kinds of startup scripts that Sage reads when starting:

2.1 The sagerc shell script

The *bash shell script* `$DOT_SAGE/sagerc` (with the default value of `DOT_SAGE`, this is `~/.sage/sagerc`) is read by `$SAGE_ROOT/spkg/bin/sage-env` after Sage has set its environment variables. It can be used to override some of the environment variables determined by Sage, or it can contain other shell commands like creating directories. This script is sourced not only when running Sage itself, but also when running any of the subcommands (like `sage --python`, `sage -b` or `sage -i <package>`). In particular, setting `PS1` here overrides the default prompt for the Sage shell `sage --sh`.

Note: This script is run with the Sage directories in its `PATH`, so executing `git` for example will run the Git inside Sage.

The default location of this file can be changed using the environment variable `SAGE_RC_FILE`.

2.2 The init.sage script

The *Sage script* `$DOT_SAGE/init.sage` (with the default value of `DOT_SAGE`, this is `~/.sage/init.sage`) contains Sage commands to be executed every time Sage starts. If you want symbolic variables `y` and `z` in every Sage session, you could put

```
var('y, z')
```

in this file.

The default location of this file can be changed using the environment variable `SAGE_STARTUP_FILE`.

CHAPTER

THREE

ENVIRONMENT VARIABLES USED BY SAGE

Sage uses several environment variables when running. These all have sensible default values, so many users won't need to set any of these. (There are also variables used to compile Sage; see the Sage Installation Guide for more about those.)

- `DOT_SAGE` – this is the directory, to which the user has read and write access, where Sage stores a number of files. The default location is `~/.sage/`, but you can change that by setting this variable.
- `SAGE_RC_FILE` – a shell script which is sourced after Sage has determined its environment variables. This script is executed before starting Sage or any of its subcommands (like `sage -i <package>`). The default value is `$DOT_SAGE/sagerc`.
- `SAGE_STARTUP_FILE` – a file including commands to be executed every time Sage starts. The default value is `$DOT_SAGE/init.sage`.
- `SAGE_SERVER` – if you want to install a Sage package using `sage -i PKG_NAME`, Sage downloads the file from the web, using the address `http://www.sagemath.org/` by default, or the address given by `SAGE_SERVER` if it is set. If you wish to set up your own server, then note that Sage will search the directories `SAGE_SERVER/packages/standard/`, `SAGE_SERVER/packages/optional/`, `SAGE_SERVER/packages/experimental/`, and `SAGE_SERVER/packages/archive/` for packages. See the script `$SAGE_ROOT/spkg/bin/sage-spkg` for the implementation.
- `SAGE_PATH` – a colon-separated list of directories which Sage searches when trying to locate Python libraries.
- `SAGE_BROWSER` – on most platforms, Sage will detect the command to run a web browser, but if this doesn't seem to work on your machine, set this variable to the appropriate command.
- `SAGE_ORIG_LD_LIBRARY_PATH_SET` – set this to something non-empty to force Sage to set the `LD_LIBRARY_PATH` before executing system commands.
- `SAGE_ORIG_DYLD_LIBRARY_PATH_SET` – similar, but only used on Mac OS X to set the `DYLD_LIBRARY_PATH`.
- `SAGE_CBLAS` – used in the file `SAGE_ROOT/devel/sage/sage/misc/cython.py`. Set this to the base name of the BLAS library file on your system if you want to override the default setting. That is, if the relevant file is called `libcblas_new.so` or `libcblas_new.dylib`, then set this to "cblas_new".

CHAPTER

FOUR

INTERACTIVELY TRACING EXECUTION OF A COMMAND

sage.misc.trace.**trace**(*code*, *preparse=True*)

Evaluate Sage code using the interactive tracer and return the result. The string `code` must be a valid expression enclosed in quotes (no assignments - the result of the expression is returned). In the Sage notebook this just raises a NotImplementedException.

INPUT:

- `code` - str
- `preparse` - bool (default: True); if True, run expression through the Sage preparser.

REMARKS: This function is extremely powerful! For example, if you want to step through each line of execution of, e.g., `factor(100)`, type

```
sage: trace("factor(100)")              # not tested
```

then at the (Pdb) prompt type `s` (or `step`), then press return over and over to step through every line of Python that is called in the course of the above computation. Type `?` at any time for help on how to use the debugger (e.g., `l` lists 11 lines around the current line; `bt` gives a back trace, etc.).

Setting a break point: If you have some code in a file and would like to drop into the debugger at a given point, put the following code at that point in the file:

```
import pdb; pdb.set_trace()
```

For an article on how to use the Python debugger, see http://www.onlamp.com/pub/a/python/2005/09/01/debugger.html

TESTS: The only real way to test this is via pexpect spawning a sage subprocess that uses IPython.

```
sage: import pexpect
sage: s = pexpect.spawn('sage')
sage: _ = s.sendline("trace('print factor(10)'); print 3+97")
sage: _ = s.sendline("s"); _ = s.sendline("c");
sage: _ = s.expect('100', timeout=90)
```

Seeing the ipdb prompt and the 2 * 5 in the output below is a strong indication that the trace command worked correctly.

```
sage: print s.before[s.before.find('--'):]
--...
ipdb> c
2 * 5
```

We test what happens in notebook embedded mode:

```
sage: sage.plot.plot.EMBEDDED_MODE = True
sage: trace('print factor(10)')
Traceback (most recent call last):
...
NotImplementedError: the trace command is not implemented in the Sage notebook; you must use the
```

CHAPTER

FIVE

EXTRA READLINE COMMANDS

Extra readline commands

The following extra readline commands are available in Sage:

- `operate-and-get-next`
- `history-search-backward-and-save`
- `history-search-forward-and-save`

The `operate-and-get-next` command accepts the input line and fetches the next line from the history. This is the same command with the same name in the Bash shell.

The `history-search-backward-and-save` command searches backward in the history for the string of characters from the start of the input line to the current cursor position, and fetches the first line found. If the cursor is at the start of the line, the previous line is fetched. The position of the fetched line is saved internally, and the next search begins at the saved position.

The `history-search-forward-and-save` command behaves similarly but forward.

The previous two commands is best used in tandem to fetch a block of lines from the history, by searching backward the first line of the block and then issuing the forward command as many times as needed. They are intended to replace the `history-search-backward` command and the `history-search-forward` command provided by the GNU readline library used in Sage.

To bind these commands with keys, insert the relevant lines into the IPython configuration file `$DOT_SAGE/ipython-*/profile_sage/ipython_config.py`. Note that `$DOT_SAGE` is `$HOME/.sage` by default. For example,

```
c = get_config()

c.InteractiveShell.readline_parse_and_bind = [
    '"\C-o": operate-and-get-next',
    '"\e[A": history-search-backward-and-save',
    '"\e[B": history-search-forward-and-save'
    ]
```

binds the three commands with the control-o key, the up arrow key, and the down arrow key, respectively. *Warning:* Sometimes, these keys may be bound to do other actions by the terminal and does not reach to the readline properly (check this by running `stty -a` and reading the `cchars` section). Then you may need to turn off these bindings before the new readline commands work fine . A prominent case is when control-o is bound to `discard` by the terminal. You can turn this off by running `stty discard undef`.

AUTHORS:

- Kwankyu Lee (2010-11-23): initial version

- Kwankyu Lee (2013-06-05): updated for the new IPython configuration format.

CHAPTER

SIX

SAGE'S IPYTHON MODIFICATIONS

This module contains all of Sage's customizations to the IPython interpreter. These changes consist of the following major components:

- `SageTerminalApp`
- `SageInteractiveShell`
- `interface_shell_embed()`

6.1 SageTerminalApp

This is the main application object. It is used by the `$SAGE_LOCAL/bin/sage-ipython` script to start the Sage command-line. It's primary purpose is to

- Initialize the `SageInteractiveShell`.
- Provide default configuration options for the shell, and its subcomponents. These work with (and can be overridden by) IPython's configuration system.
- Load the Sage ipython extension (which does things like preparsing, add magics, etc.).
- Provide a custom `SageCrashHandler` to give the user instructions on how to report the crash to the Sage support mailing list.

6.2 SageInteractiveShell

The `SageInteractiveShell` object is the object responsible for accepting input from the user and evaluating it. From the command-line, this object can be retrieved by running:

```
sage: shell = get_ipython()    # not tested
```

The `SageInteractiveShell` provides the following customizations:

- Modify the libraries before calling system commands. See `system_raw()`.

6.3 Interface Shell

The function `interface_shell_embed()` takes a `Interface` object and returns an embeddable IPython shell which can be used to directly interact with that shell. The bulk of this functionality is provided through

InterfaceShellTransformer.

class sage.repl.interpreter.**InterfaceShellTransformer**(**args*, ***kwds*)

Bases: IPython.core.prefilter.PrefilterTransformer

Initialize this class. All of the arguments get passed to PrefilterTransformer.__init__().

temporary_objects

a list of hold onto interface objects and keep them from being garbage collected

See Also:

interface_shell_embed()

EXAMPLES:

```
sage: from sage.repl.interpreter import interface_shell_embed
sage: shell = interface_shell_embed(maxima)
sage: ift = shell.prefilter_manager.transformers[0]
sage: ift.temporary_objects
set([])
sage: ift._sage_import_re.findall('sage(a) + maxima(b)')
['a', 'b']
```

preparse_imports_from_sage(*line*)

Finds occurrences of strings such as sage(object) in *line*, converts object to shell.interface, and replaces those strings with their identifier in the new system. This also works with strings such as maxima(object) if shell.interface is maxima.

Parameters **line** (*string*) – the line to transform

Warning: This does not parse nested parentheses correctly. Thus, lines like sage(a.foo()) will not work correctly. This can't be done in generality with regular expressions.

EXAMPLES:

```
sage: from sage.repl.interpreter import interface_shell_embed, InterfaceShellTransformer
sage: shell = interface_shell_embed(maxima)
sage: ift = InterfaceShellTransformer(shell=shell, config=shell.config, prefilter_manager=sh
sage: ift.shell.ex('a = 3')
sage: ift.preparse_imports_from_sage('2 + sage(a)')
'2 + sage0 '
sage: maxima.eval('sage0')
'3'
sage: ift.preparse_imports_from_sage('2 + maxima(a)')
'2 +  sage1 '
sage: ift.preparse_imports_from_sage('2 + gap(a)')
'2 + gap(a)'
```

transform(*line*, *continue_prompt*)

Evaluates *line* in shell.interface and returns a string representing the result of that evaluation.

Parameters

- **line** (*string*) – the line to be transformed *and evaluated*
- **continue_prompt** (*bool*) – is this line a continuation in a sequence of multiline input?

EXAMPLES:

```
sage: from sage.repl.interpreter import interface_shell_embed, InterfaceShellTransformer
sage: shell = interface_shell_embed(maxima)
sage: ift = InterfaceShellTransformer(shell=shell, config=shell.config, prefilter_manager=sh
```

```
sage: ift.transform('2+2', False)    # note: output contains triple quotation marks
'sage.misc.all.logstr("""4""")'
sage: ift.shell.ex('a = 4')
sage: ift.transform(r'sage(a)+4', False)
'sage.misc.all.logstr("""8""")'
sage: ift.temporary_objects
set([])
sage: shell = interface_shell_embed(gap)
sage: ift = InterfaceShellTransformer(shell=shell, config=shell.config, prefilter_manager=sh
sage: ift.transform('2+2', False)
'sage.misc.all.logstr("""4""")'
```

class sage.repl.interpreter.**SageCrashHandler**(*app*)

Bases: IPython.terminal.ipapp.IPAppCrashHandler

A custom CrashHandler which gives the user instructions on how to post the problem to sage-support.

EXAMPLES:

```
sage: from sage.repl.interpreter import SageTerminalApp, SageCrashHandler
sage: app = SageTerminalApp.instance()
sage: sch = SageCrashHandler(app); sch
<sage.repl.interpreter.SageCrashHandler object at 0x...>
sage: sorted(sch.info.items())
[('app_name', u'Sage'),
 ('bug_tracker', 'http://trac.sagemath.org'),
 ('contact_email', 'sage-support@googlegroups.com'),
 ('contact_name', 'sage-support'),
 ('crash_report_fname', u'Crash_report_Sage.txt')]
```

class sage.repl.interpreter.**SageInteractiveShell**(*config=None*, *ipython_dir=None*, *profile_dir=None*, *user_ns=None*, *user_module=None*, *custom_exceptions=((), None)*, *usage=None*, *banner1=None*, *banner2=None*, *display_banner=None*, ***kwargs*)

Bases: IPython.terminal.interactiveshell.TerminalInteractiveShell

system_raw(*cmd*)

Run a system command.

If the command is not a sage-specific binary, adjust the library paths before calling system commands. See trac ticket #975 for a discussion of running system commands.

This is equivalent to the sage-native-execute shell script.

EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.system_raw('false')
sage: shell.user_ns['_exit_code'] > 0
True
sage: shell.system_raw('true')
sage: shell.user_ns['_exit_code']
0
sage: shell.system_raw('env | grep "^LD_LIBRARY_PATH=" | grep $SAGE_LOCAL')
sage: shell.user_ns['_exit_code']
1
```

```
sage: shell.system_raw('R --version')
R version ...
sage: shell.user_ns['_exit_code']
0
```

sage.repl.interpreter.**SagePreparseTransformer**(***kwargs*)

EXAMPLES:

```
sage: from sage.repl.interpreter import SagePreparseTransformer
sage: spt = SagePreparseTransformer()
sage: spt.push('1+1r+2.3^2.3r')
"Integer(1)+1+RealNumber('2.3')**2.3"
sage: preparser(False)
sage: spt.push('2.3^2')
'2.3^2'
```

TESTS:

Check that syntax errors in the preparser do not crash IPython, see trac ticket #14961.

```
sage: preparser(True)
sage: bad_syntax = "R.<t> = QQ[]"
sage: preparse(bad_syntax)
Traceback (most recent call last):
...
SyntaxError: Mismatched ']'
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell(bad_syntax)
  File "<string>", line unknown
SyntaxError: Mismatched ']'
```

sage.repl.interpreter.**SagePromptTransformer**(***kwargs*)

Strip the sage:/....: prompts of Sage.

EXAMPLES:

```
sage: from sage.repl.interpreter import SagePromptTransformer
sage: spt = SagePromptTransformer()
sage: spt.push("sage: 2 + 2")
'2 + 2'
sage: spt.push('')
''
sage: spt.push("....: 2+2")
'2+2'
```

This should strip multiple prompts: see trac ticket #16297:

```
sage: spt.push("sage:    sage: 2+2")
'2+2'
sage: spt.push("    sage: ....: 2+2")
'2+2'
```

The prompt contains a trailing space. Extra spaces between the last prompt and the remainder should not be stripped:

```
sage: spt.push("    sage: ....:     2+2")
'    2+2'
```

We test that the input transformer is enabled on the Sage command line:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('sage: a = 123')              # single line
sage: shell.run_cell('sage: a = [\n... 123]')      # old-style multi-line
sage: shell.run_cell('sage: a = [\n....: 123]')    # new-style multi-line
```

We test that trac ticket #16196 is resolved:

```
sage: shell.run_cell('    sage: 1+1')
2
```

class sage.repl.interpreter.**SageTerminalApp**(***kwargs*)

Bases: IPython.terminal.ipapp.TerminalIPythonApp

crash_handler_class

alias of SageCrashHandler

init_shell()

Initialize the SageInteractiveShell instance.

Note: This code is based on TermintalIPythonApp.init_shell().

EXAMPLES:

```
sage: from sage.repl.interpreter import SageTerminalApp, DEFAULT_SAGE_CONFIG
sage: app = SageTerminalApp(config=DEFAULT_SAGE_CONFIG)
sage: app.initialize(argv=[])  # indirect doctest
sage: app.shell
<sage.repl.interpreter.SageInteractiveShell object at 0x...>
```

load_config_file(**args*, ***kwds*)

Merges a config file with the default sage config.

Note: This code is based on Application.update_config().

TESTS:

Test that trac ticket #15972 has been fixed:

```
sage: from sage.misc.temporary_file import tmp_dir
sage: from sage.repl.interpreter import SageTerminalApp
sage: d = tmp_dir()
sage: IPYTHONDIR = os.environ['IPYTHONDIR']
sage: os.environ['IPYTHONDIR'] = d
sage: SageTerminalApp().load_config_file()
sage: os.environ['IPYTHONDIR'] = IPYTHONDIR
```

sage.repl.interpreter.**embedded**()

Returns True if Sage is being run from the notebook.

EXAMPLES:

```
sage: from sage.repl.interpreter import embedded
sage: embedded()
False
```

sage.repl.interpreter.**get_test_shell**()

Returns a IPython shell that can be used in testing the functions in this module.

Returns an IPython shell

EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell(); shell
<sage.repl.interpreter.SageInteractiveShell object at 0x...>
```

TESTS:

Check that trac ticket #14070 has been resolved:

```
sage: from sage.tests.cmdline import test_executable
sage: cmd = 'from sage.repl.interpreter import get_test_shell; shell = get_test_shell()'
sage: (out, err, ret) = test_executable(["sage", "-c", cmd])
sage: out + err
''
```

sage.repl.interpreter.**interface_shell_embed**(*interface*)

Returns an IPython shell which uses a Sage interface on the backend to perform the evaluations. It uses `InterfaceShellTransformer` to transform the input into the appropriate `interface.eval(...)` input.

INPUT:

- `interface` – A Sage `PExpect` interface instance.

EXAMPLES:

```
sage: from sage.repl.interpreter import interface_shell_embed
sage: shell = interface_shell_embed(gap)
sage: shell.run_cell('List( [1..10], IsPrime )')
[ false, true, true, false, true, false, true, false, false, false ]
```

sage.repl.interpreter.**preparser**(*on=True*)

Turn on or off the Sage preparser.

Parameters **on** (*bool*) – if True turn on preparsing; if False, turn it off.

EXAMPLES:

```
sage: 2/3
2/3
sage: preparser(False)
sage: 2/3  # not tested since doctests are always preparsed
0
sage: preparser(True)
sage: 2^3
8
```

CHAPTER

SEVEN

SAGE'S IPYTHON EXTENSION

A Sage extension which adds sage-specific features:

- magics
  - `%crun`
  - `%runfile`
  - `%attach`
  - `%display`
  - `%mode` (like `%maxima`, etc.)
- preparsing of input
- loading Sage library
- running init.sage
- changing prompt to Sage prompt
- Display hook

TESTS:

We test that preparsing is off for `%runfile`, on for `%time`:

```
sage: import os, re
sage: from sage.repl.interpreter import get_test_shell
sage: from sage.misc.misc import tmp_dir
sage: shell = get_test_shell()
sage: TMP = tmp_dir()
```

The temporary directory should have a name of the form `.../12345/...`, to demonstrate that file names are not preparsed when calling `%runfile`

```
sage: bool(re.search('/[0-9]+/', TMP))
True
sage: tmp = os.path.join(TMP, 'run_cell.py')
sage: f = open(tmp, 'w'); f.write('a = 2\n'); f.close()
sage: shell.run_cell('%runfile '+tmp)
sage: shell.run_cell('a')
2
```

In contrast, input to the `%time` magic command is preparsed:

```
sage: shell.run_cell('%time 594.factor()')
CPU times: user ...
Wall time: ...
2 * 3^3 * 11
```

class sage.repl.ipython_extension.**SageCustomizations**(*shell=None*)
Bases: object

Initialize the Sage plugin.

init_environment()
Set up Sage command-line environment

init_inspector()
x.__init__(...) initializes x; see help(type(x)) for signature

init_line_transforms()
Set up transforms (like the preparser).

register_interface_magics()
Register magics for each of the Sage interfaces

run_init()
Run Sage's initial startup file.

set_quit_hook()
Set the exit hook to cleanly exit Sage.

class sage.repl.ipython_extension.**SageMagics**(*shell=None*, ***kwargs*)
Bases: IPython.core.magic.Magics

attach(*s*)
Attach the code contained in the file s.

This is designed to be used from the command line as %attach /path/to/file.

- s – string. The file to be attached

EXAMPLES:

```
sage: import os
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: tmp = os.path.normpath(os.path.join(SAGE_TMP, 'run_cell.py'))
sage: f = open(tmp, 'w'); f.write('a = 2\n'); f.close()
sage: shell.run_cell('%attach ' + tmp)
sage: shell.run_cell('a')
2
sage: sleep(1)  # filesystem timestamp granularity
sage: f = open(tmp, 'w'); f.write('a = 3\n'); f.close()
```

Note that the doctests are never really at the command prompt, so we call the input hook manually:

```
sage: shell.run_cell('from sage.repl.inputhook import sage_inputhook')
sage: shell.run_cell('sage_inputhook()')
### reloading attached file run_cell.py modified at ... ###
0

sage: shell.run_cell('a')
3
sage: shell.run_cell('detach(%r)'%tmp)
sage: shell.run_cell('attached_files()')
```

```
[]
sage: os.remove(tmp)
```

crun(*s*)

Profile C function calls

INPUT:

- s – string. Sage command to profile.

EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('%crun sum(1/(1+n^2) for n in range(100))')    # optional - gperftools
PROFILE: interrupts/evictions/bytes = ...
Using local file ...
Using local file ...
```

display(*args*)

A magic command to switch between simple display and ASCII art display.

- args – string. See `sage.misc.display_hook.DisplayHookBase.set_display()` for allowed values. If the mode is `ascii_art`, it can optionally be followed by a width.

How to use: if you want activate the ASCII art mod:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('%display ascii_art')
```

That means you don't have to use `ascii_art()` to get an ASCII art output:

```
sage: shell.run_cell("i = var('i')")
sage: shell.run_cell('sum(i^2*x^i, i, 0, 10)')
     10       9       8       7       6       5       4      3      2
100*x   + 81*x  + 64*x  + 49*x  + 36*x  + 25*x  + 16*x  + 9*x  + 4*x  + x
```

Then when you want return in 'textual mode':

```
sage: shell.run_cell('%display simple')
sage: shell.run_cell('sum(i^2*x^i, i, 0, 10)')
100*x^10 + 81*x^9 + 64*x^8 + 49*x^7 + 36*x^6 + 25*x^5 + 16*x^4 + 9*x^3 + 4*x^2 + x
```

Sometime you could have to use a special output width and you could specify it:

```
sage: shell.run_cell('%display ascii_art')
sage: shell.run_cell('StandardTableaux(4).list()')
[
[                                                                 1  4    1  3
[                 1  3  4    1  2  4    1  2  3    1  3    1  2    2       2
[   1  2  3  4,   2     ,   3     ,   4     ,   2  4,   3  4,   3   ,   4

            1 ]
    1  2    2 ]
    3       3 ]
,   4   ,   4 ]
sage: shell.run_cell('%display ascii_art 50')
sage: shell.run_cell('StandardTableaux(4).list()')
[
[
```

```
[                 1  3  4    1  2  4    1  2  3
[   1  2  3  4,   2     ,   3     ,   4     ,

                                          1 ]
                  1  4    1  3    1  2    2 ]
  1  3    1  2    2       2       3       3 ]
  2  4,   3  4,   3   ,   4   ,   4   ,   4 ]
```

As yet another option, typeset mode. This is used in the emacs interface:

```
sage: shell.run_cell('%display typeset')
sage: shell.run_cell('1/2')
<html><script type="math/tex">...\frac{1}{2}</script></html>
```

Switch back:

```
sage: shell.run_cell('%display simple')
```

TESTS:

```
sage: shell.run_cell('%display invalid_mode')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
...
ValueError: invalid mode set
```

iload(*args*)

A magic command to interactively load a file as in MAGMA.

- `args` – string. The file to be interactively loaded

Note: Currently, this cannot be completely doctested as it relies on `raw_input()`.

EXAMPLES:

```
sage: ip = get_ipython()           # not tested: works only in interactive shell
sage: ip.magic_iload('/dev/null')  # not tested: works only in interactive shell
Interactively loading "/dev/null"  # not tested: works only in interactive shell
```

runfile(*s*)

Execute the code contained in the file `s`.

This is designed to be used from the command line as `%runfile /path/to/file`.

- `s` – string. The file to be loaded.

EXAMPLES:

```
sage: import os
sage: from sage.repl.interpreter import get_test_shell
sage: from sage.misc.misc import tmp_dir
sage: shell = get_test_shell()
sage: tmp = os.path.join(tmp_dir(), 'run_cell.py')
sage: f = open(tmp, 'w'); f.write('a = 2\n'); f.close()
sage: shell.run_cell('%runfile '+tmp)
sage: shell.run_cell('a')
2
```

sage.repl.ipython_extension.**load_ipython_extension**(**args*, ***kwargs*)

Load the extension in IPython.

sage.repl.ipython_extension.**run_once**(*func*)

Runs a function (successfully) only once.

The running can be reset by setting the `has_run` attribute to False

TEST:

```
sage: from sage.repl.ipython_extension import run_once
sage: @run_once
....: def foo(work):
....:     if work:
....:         return 'foo worked'
....:     raise RuntimeError("foo didn't work")
sage: foo(False)
Traceback (most recent call last):
...
RuntimeError: foo didn't work
sage: foo(True)
'foo worked'
sage: foo(False)
sage: foo(True)
```

For more details about how to use the Sage command line once it starts up, see the Sage tutorial and the documentation for IPython.

CHAPTER

EIGHT

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

m

r

INDEX

Symbols

A

C

D

E

G

I

L

P

R

S

T