The Sage Command Line

Release 10.6

The Sage Development Team

Apr 01, 2025

CONTENTS

The Sage Read-Eval-Print-Loop (REPL) is based on IPython. In this document, you'll find how the IPython integration works. You should also be familiar with the documentation for IPython.

For more details about using the Sage command line, see the Sage tutorial.

RUNNING SAGE

1.1 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.

1.1.1 Command-line options for Sage

```
SageMath version 10.6, Release Date: 2025-03-31

Running Sage, the most common options:

  file.[sage|py|spyx] -- run given .sage, .py or .spyx file
  -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", and produce
                          the corresponding Python file
                          "file.sage.py"
  -q                  -- quiet; start with no banner
  --min               -- do not populate global namespace
                         (must be first option)
  --nodotsage         -- run Sage without using the user's
                          .sage directory: create and use a temporary
                          .sage directory instead.
  --gthread, --qthread, --q4thread, --wthread, --pylab
                      -- pass the option through to IPython
  --simple-prompt     -- pass the option through to IPython: use
                         this option with sage-shell mode in emacs
  --gdb               -- run Sage under the control of gdb
  --lldb              -- run Sage under the control of lldb

Running external programs:

  --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
```

(continues on next page)

```
  --gap3 [...]          -- run Sage's Gap3 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 [...], --ipython3 [...]
                        -- run Sage's IPython using the default
                           environment (not Sage), passing additional
                           additional options to IPython
  --jupyter [...]       -- run Sage's Jupyter with given arguments
  --kash [...]          -- run Sage's Kash with the given arguments
                           (not installed currently, run sage -i kash)
  --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
  --pip [...]           -- invoke pip, the Python package manager
  --polymake [...]      -- run Sage's Polymake with given arguments
  --python [...], --python3 [...]
                        -- run the Python 3 interpreter
  -R [...]              -- run Sage's R with the given arguments
  --singular [...]      -- run Sage's singular with the given arguments
  --sqlite3 [...]       -- run Sage's sqlite3 with given arguments

Running the notebook:

  -n [...], --notebook=[...]
                        -- start the notebook; valid options include
                           'default', 'jupyter', 'jupyterlab', and 'export'.
                           Current default is 'jupyter'.
                           Run "sage --notebook --help" for more details.

Testing files:

  -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
    --all             -- test all files
    --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 block
                         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
    --random-seed[=seed]  -- random seed (integer) for fuzzing doctests
    --new             -- only test files modified since last commit
    --initial         -- only show the first failure per block
```

```
    --debug          -- drop into PDB after an unexpected error
    --failed         -- only test files that failed last test
    --warn-long [timeout] -- warn if tests take too much CPU time
    --only-errors    -- only output failures, not successes
    --gc=GC          -- control garbage collection (ALWAYS:
                        collect garbage before every test; NEVER:
                        disable gc; DEFAULT: Python default)
    --short[=secs]   -- run as many doctests as possible in about 300
                        seconds (or the number of seconds given.) This runs
                        the tests for each module from the top of the file
                        and skips tests once it exceeds the budget
                        allocated for that file.
    --help           -- show all doctesting options
 --tnew [...]        -- equivalent to -t --new
 -tp <N> [...]       -- like -t above, but tests in parallel using
                        N threads, with 0 interpreted as min(8, cpu_count())
 --testall [options] -- equivalent to -t --all

 --coverage <files>  -- give information about doctest coverage of files
 --coverageall       -- give summary info about doctest coverage of
                        all files in the Sage library
 --startuptime [module] -- display how long each component of Sage takes to
                        start up; optionally specify a module to get more
                        details about that particular module
 --tox [options] <files|dirs> -- general entry point for testing
                                and linting of the Sage library
    -e <envlist>     -- run specific test environments; default:
                        doctest,coverage,startuptime,pycodestyle-minimal,relint,
↪codespell,rst,ruff-minimal
        doctest              -- run the Sage doctester
                                (same as "sage -t")
        coverage             -- give information about doctest coverage of files
                                (same as "sage --coverage[all]")
        startuptime          -- display how long each component of Sage takes to↪
↪start up
                                (same as "sage --startuptime")
        pycodestyle-minimal  -- check against Sage's minimal style conventions
        relint               -- check whether some forbidden patterns appear
        codespell            -- check for misspelled words in source code
        rst                  -- validate Python docstrings markup as↪
↪reStructuredText
        ruff-minimal         -- check against Sage's minimal style conventions
        coverage.py          -- run the Sage doctester with Coverage.py
        coverage.py-html     -- run the Sage doctester with Coverage.py, generate↪
↪HTML report
        coverage.py-xml      -- run the Sage doctester with Coverage.py, generate↪
↪XML report
        pyright              -- run the static typing checker pyright
        pycodestyle          -- check against the Python style conventions of PEP8
        cython-lint          -- check Cython files for code style
        ruff                 -- check against Python style conventions
    -p auto          -- run test environments in parallel
```

```
    --help           -- show tox help
  --pytest [options] <files|dirs> -- run pytest on the Sage library
    --help           -- show pytest help


Some developer utilities:

  --grep [options] <string>
                     -- regular expression search through the Sage
                        library for "string". Any options will
                        get passed to the "grep" command.
  --grepdoc [options] <string>
                     -- regular expression search through the
                        Sage documentation for "string".
  --search_src ...   -- same as --grep
  --search_doc ...   -- same as --grepdoc
  --fixdoctests file.py
                     -- Run doctests and replace output of failing doctests
                        with actual output.
  --fixdistributions <files|dirs>
                     -- Check or update '# sage_setup: distribution'
                        directives in source files
  --sh [...]         -- run a shell with Sage environment variables
                        as they are set in the runtime of Sage
  --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.)
File conversion:

  --rst2ipynb [...]  -- Generates Jupyter notebook (.ipynb) from standalone
                        reStructuredText source.
                        (not installed currently, run sage -i rst2ipynb)
  --ipynb2rst [...]  -- Generates a reStructuredText source file from
                        a Jupyter notebook (.ipynb).
  --sws2rst <sws doc> -- Generates a reStructuredText source file from
                        a Sage worksheet (.sws) document.
                        (not installed currently, run sage -i sage_sws2rst)


Valgrind memory debugging:

  --cachegrind       -- run Sage using Valgrind's cachegrind tool.  The log
                        files are named sage-cachegrind.PID can be found in
                        $DOT_SAGE
  --callgrind        -- run Sage using Valgrind's callgrind tool.  The log
                        files are named sage-callgrind.PID can be found in
                        $DOT_SAGE
  --massif           -- run Sage using Valgrind's massif tool.  The log
                        files are named sage-massif.PID can be found in
                        $DOT_SAGE
  --memcheck         -- run Sage using Valgrind's memcheck tool.  The log
                        files are named sage-memcheck.PID can be found in
                        $DOT_SAGE
```

```
  --omega            -- run Sage using Valgrind's omega tool.  The log
                        files are named sage-omega.PID can be found in
                        $DOT_SAGE
  --valgrind         -- this is an alias for --memcheck

Getting help:

  -v, --version      -- display Sage version information
  --dumpversion      -- print brief Sage version
  -h, -?, --help     -- print a short help message
  --advanced         -- list all command line options

Building the Sage library:

  -b                 -- build Sage library -- do this if you have
                        modified any source code files in SAGE_ROOT/src/sage/
  -ba                -- same as -b, but rebuild *all* Cython
                        code.
  -br                -- build and run Sage

  -bt [...]          -- build Sage and test; same options as -t
  -btp <N> [...]     -- build Sage and test in parallel; same options as -tp
  -btnew [...]       -- build Sage and test modified files, as in -t --new

  -bn [...], --build-and-notebook [...]
                     -- build the Sage library (as by running "sage -b")
                        and then start the notebook

Package handling:

  --package [args]   -- call the package manager with given arguments.
                        Run without arguments for help.
  --experimental     -- list all experimental packages that can be installed
  -i [opts] [pkgs]   -- install the given Sage packages.  Options:
                            -c -- run the packages' test suites,
                                  overriding the settings of
                                  SAGE_CHECK and SAGE_CHECK_PACKAGES
                            -d -- only download, do not install packages
                            -f -- force build: install the packages even
                                  if they are already installed
                            -s -- do not delete the temporary build directories
                                  after a successful build
                            -y -- reply yes to prompts about experimental
                                  and old-style packages; warning: there
                                  is no guarantee that these packages will
                                  build correctly; use at your own risk
                            -n -- reply no to prompts about experimental
                                  and old-style packages
  -f [opts] [pkgs]   -- shortcut for -i -f: force build of the given Sage
                        packages
  -p [opts] [packages]-- install the given Sage packages, without dependency
                        checking. Options are the same as for the -i command.
```

```
  --optional          -- list all optional packages that can be installed
  --standard          -- list all standard packages that can be installed
  --installed         -- list all installed packages

Making Sage distributions:

  --sdist             -- build a source distribution of Sage

Building the documentation:

  --docbuild [lang/]<document> <html|pdf|...> -- Build the Sage documentation

Other developer tools:

  --root              -- print the Sage root directory
  --git-branch        -- print the current git branch
  --buildsh [...]     -- run a shell with Sage environment variables
                         as they are set while building Sage and its packages
```

1.2 Sage startup scripts

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

1.2.1 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/src/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 shells `sage --buildsh` and `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`.

1.2.2 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`.

1.3 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` – only used for installing packages. Alternative mirror from which to download sources, see the Installation Guide for details.

- `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.

1.4 Relevant environment variables for other packages

This is a non-exhaustive list of environment variables which influence some package contained within the SageMath distribution.

In many cases, SageMath uses a custom default value if the variable is not set, which is not the same default that the system-wide package would use. So, if you would like to use your system-wide configuration, you need to explicitly set the environment variable to the system-wide default.

- `IPYTHONDIR` – directory where the configuration of IPython is stored. By default, this is some directory inside `DOT_SAGE`. See http://ipython.readthedocs.io/en/stable/development/config.html for more information.

- `JUPYTER_CONFIG_DIR` – directory where the configuration of Jupyter is stored. By default, this is some directory inside `DOT_SAGE`. See http://jupyter.readthedocs.io/en/latest/projects/jupyter-directories.html for more information.

- `MPLCONFIGDIR` – directory where the configuration of Matplotlib is stored. See https://matplotlib.org/faq/environment_variables_faq.html#envvar-MPLCONFIGDIR By default, this is some directory inside `DOT_SAGE`.

1.5 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` – string

- `preparse` – boolean (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: from sage.misc.trace import trace
sage: trace("factor(100)")              # not tested
```

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

PREPARSING

Sage commands are "preparsed" to valid Python syntax. This allows for example to support the `R.<x> = QQ[]` syntax.

2.1 The Sage Preparser

EXAMPLES:

Preparsing:

```
sage: preparse('2/3')
'Integer(2)/Integer(3)'
sage: preparse('2.5')
"RealNumber('2.5')"
sage: preparse('2^3')
'Integer(2)**Integer(3)'
sage: preparse('a^b')              # exponent
'a**b'
sage: preparse('a**b')
'a**b'
sage: preparse('G.0')              # generator
'G.gen(0)'
sage: preparse('a = 939393R')      # raw
'a = 939393'
sage: implicit_multiplication(True)
sage: preparse('a b c in L')       # implicit multiplication
'a*b*c in L'
sage: preparse('2e3x + 3exp(y)')
"RealNumber('2e3')*x + Integer(3)*exp(y)"
```

```
>>> from sage.all import *
>>> preparse('2/3')
'Integer(2)/Integer(3)'
>>> preparse('2.5')
"RealNumber('2.5')"
>>> preparse('2^3')
'Integer(2)**Integer(3)'
>>> preparse('a^b')              # exponent
'a**b'
>>> preparse('a**b')
'a**b'
>>> preparse('G.0')              # generator
```

```
'G.gen(0)'
>>> preparse('a = 939393R')     # raw
'a = 939393'
>>> implicit_multiplication(True)
>>> preparse('a b c in L')      # implicit multiplication
'a*b*c in L'
>>> preparse('2e3x + 3exp(y)')
"RealNumber('2e3')*x + Integer(3)*exp(y)"
```

A string with escaped quotes in it (the point here is that the preparser does not get confused by the internal quotes):

```
sage: ""Yes," he said."
'"Yes," he said.'
sage: s = "\"; s
'\'
```

```
>>> from sage.all import *
>>> ""Yes," he said."
'"Yes," he said.'
>>> s = "\"; s
'\'
```

A hex literal:

```
sage: preparse('0x2e3')
'Integer(0x2e3)'
sage: 0xA
10
sage: 0xe
14
```

```
>>> from sage.all import *
>>> preparse('0x2e3')
'Integer(0x2e3)'
>>> Integer(0xA)
10
>>> Integer(0xe)
14
```

Raw and hex work correctly:

```
sage: type(0xa1)
<class 'sage.rings.integer.Integer'>
sage: type(0xa1r)
<class 'int'>
sage: type(0Xa1R)
<class 'int'>
```

```
>>> from sage.all import *
>>> type(Integer(0xa1))
<class 'sage.rings.integer.Integer'>
>>> type(0xa1)
```

```
<class 'int'>
>>> type(0Xa1)
<class 'int'>
```

The preparser can handle PEP 515 (see Issue #28490):

```
sage: 1_000_000 + 3_000
1003000
```

```
>>> from sage.all import *
>>> Integer(1_000_000) + Integer(3_000)
1003000
```

In Sage, methods can also be called on integer and real literals (note that in pure Python this would be a syntax error):

```
sage: 16.sqrt()
4
sage: 87.factor()
3 * 29
sage: 15.10.sqrt()                                                      #␣
→needs sage.rings.real_mpfr
3.88587184554509
sage: preparse('87.sqrt()')
'Integer(87).sqrt()'
sage: preparse('15.10.sqrt()')
"RealNumber('15.10').sqrt()"
```

```
>>> from sage.all import *
>>> Integer(16).sqrt()
4
>>> Integer(87).factor()
3 * 29
>>> RealNumber('15.10').sqrt()                                          ␣
→           # needs sage.rings.real_mpfr
3.88587184554509
>>> preparse('87.sqrt()')
'Integer(87).sqrt()'
>>> preparse('15.10.sqrt()')
"RealNumber('15.10').sqrt()"
```

Note that calling methods on int literals in pure Python is a syntax error, but Sage allows this for Sage integers and reals, because users frequently request it:

```
sage: eval('4.__add__(3)')
Traceback (most recent call last):
...
SyntaxError: invalid ...
```

```
>>> from sage.all import *
>>> eval('4.__add__(3)')
Traceback (most recent call last):
```

```
...
SyntaxError: invalid ...
```

Symbolic functional notation:

```
sage: # needs sage.symbolic
sage: a = 10; f(theta, beta) = theta + beta; b = x^2 + theta
sage: f
(theta, beta) |--> beta + theta
sage: a
10
sage: b
x^2 + theta
sage: f(theta,theta)
2*theta

sage: a = 5; f(x,y) = x*y*sqrt(a)                                      #␣
→needs sage.symbolic
sage: f                                                                #␣
→needs sage.symbolic
(x, y) |--> sqrt(5)*x*y
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> a = Integer(10); __tmp__=var("theta,beta"); f = symbolic_expression(theta + beta).
→function(theta,beta); b = x**Integer(2) + theta
>>> f
(theta, beta) |--> beta + theta
>>> a
10
>>> b
x^2 + theta
>>> f(theta,theta)
2*theta

>>> a = Integer(5); __tmp__=var("x,y"); f = symbolic_expression(x*y*sqrt(a)        ␣
→                                       ).function(x,y)# needs sage.symbolic
>>> f                                                                  #␣
→needs sage.symbolic
(x, y) |--> sqrt(5)*x*y
```

This involves an =-, but should still be turned into a symbolic expression:

```
sage: preparse('a(x) =- 5')
'__tmp__=var("x"); a = symbolic_expression(- Integer(5)).function(x)'
sage: f(x)=-x                                                          #␣
→needs sage.symbolic
sage: f(10)                                                            #␣
→needs sage.symbolic
-10
```

```
>>> from sage.all import *
>>> preparse('a(x) =- 5')
'__tmp__=var("x"); a = symbolic_expression(- Integer(5)).function(x)'
>>> __tmp__=var("x"); f = symbolic_expression(-x                              ␣
↪                                        ).function(x)# needs sage.symbolic
>>> f(Integer(10))                                                           ␣
↪        # needs sage.symbolic
-10
```

This involves -=, which should not be turned into a symbolic expression (of course a(x) is not an identifier, so this will never be valid):

```
sage: preparse('a(x) -= 5')
'a(x) -= Integer(5)'
```

```
>>> from sage.all import *
>>> preparse('a(x) -= 5')
'a(x) -= Integer(5)'
```

Raw literals:

Raw literals are not preparsed, which can be useful from an efficiency point of view. In Sage raw integer and floating literals are followed by an"r" (or "R") for raw, meaning not preparsed.

We create a raw integer:

```
sage: a = 393939r
sage: a
393939
sage: type(a)
<class 'int'>
```

```
>>> from sage.all import *
>>> a = 393939
>>> a
393939
>>> type(a)
<class 'int'>
```

We create a raw float:

```
sage: z = 1.5949r
sage: z
1.5949
sage: type(z)
<class 'float'>
```

```
>>> from sage.all import *
>>> z = 1.5949
>>> z
1.5949
>>> type(z)
<class 'float'>
```

You can also use an upper case letter:

```
sage: z = 3.1415R
sage: z
3.1415
sage: type(z)
<class 'float'>
```

```
>>> from sage.all import *
>>> z = 3.1415
>>> z
3.1415
>>> type(z)
<class 'float'>
```

This next example illustrates how raw literals can be very useful in certain cases. We make a list of even integers up to 10000:

```
sage: v = [ 2*i for i in range(10000)]
```

```
>>> from sage.all import *
>>> v = [ Integer(2)*i for i in range(Integer(10000))]
```

This takes a noticeable fraction of a second (e.g., 0.25 seconds). After preparsing, what Python is really executing is the following:

```
sage: preparse('v = [ 2*i for i in range(10000)]')
'v = [ Integer(2)*i for i in range(Integer(10000))]'
```

```
>>> from sage.all import *
>>> preparse('v = [ 2*i for i in range(10000)]')
'v = [ Integer(2)*i for i in range(Integer(10000))]'
```

If instead we use a raw 2 we get execution that is *instant* (0.00 seconds):

```
sage: v = [ 2r * i for i in range(10000r)]
```

```
>>> from sage.all import *
>>> v = [ 2 * i for i in range(10000)]
```

Behind the scenes what happens is the following:

```
sage: preparse('v = [ 2r * i for i in range(10000r)]')
'v = [ 2 * i for i in range(10000)]'
```

```
>>> from sage.all import *
>>> preparse('v = [ 2r * i for i in range(10000r)]')
'v = [ 2 * i for i in range(10000)]'
```

> ⚠ **Warning**
>
> The results of the above two expressions are different. The first one computes a list of Sage integers, whereas the

> second creates a list of Python integers. Python integers are typically much more efficient than Sage integers when they are very small; large Sage integers are much more efficient than Python integers, since they are implemented using the GMP C library.

F-Strings ([PEP 498](#)):

Expressions embedded within F-strings are preparsed:

```
sage: f'{1/3}'
'1/3'
sage: f'{2^3}'
'8'
sage: x = 20
sage: f'{x} in binary is: {x:08b}'
'20 in binary is: 00010100'
sage: f'{list(map(lambda x: x^2, [1, 2, .., 5]))}'
'[1, 4, 9, 16, 25]'
```

```
>>> from sage.all import *
>>> f'{Integer(1)/Integer(3)}'
'1/3'
>>> f'{Integer(2)**Integer(3)}'
'8'
>>> x = Integer(20)
>>> f'{x} in binary is: {x:08b}'
'20 in binary is: 00010100'
>>> f'{list(map(lambda x: x**Integer(2), (ellipsis_range(Integer(1), Integer(2),
→Ellipsis, Integer(5)))))}'
'[1, 4, 9, 16, 25]'
```

Note that the format specifier is not preparsed. Expressions within it, however, are:

```
sage: f'{x:10r}'
Traceback (most recent call last):
...
ValueError: Unknown format code 'r' for object of type 'int'
sage: f'{x:{10r}}'
'        20'
```

```
>>> from sage.all import *
>>> f'{x:10r}'
Traceback (most recent call last):
...
ValueError: Unknown format code 'r' for object of type 'int'
>>> f'{x:{10}}'
'        20'
```

Nested F-strings are also supported:

```
sage: f'{ f"{ 1/3 + 1/6 }" }'
'1/2'
sage: f'''1{ f"2{ f'4{ 2^3 }4' }2" }1'''
'1248421'
```

```
>>> from sage.all import *
>>> f'{ f"{ Integer(1)/Integer(3) + Integer(1)/Integer(6) }" }'
'1/2'
>>> f'''1{ f"2{ f'4{ Integer(2)**Integer(3) }4' }2" }1'''
'1248421'
```

AUTHORS:

- William Stein (2006-02-19): fixed bug when loading .py files

- William Stein (2006-03-09): fixed crash in parsing exponentials; precision of real literals now determined by digits of input (like Mathematica)

- Joe Wetherell (2006-04-14): added MAGMA-style constructor preparsing

- Bobby Moretti (2007-01-25): added preliminary function assignment notation

- Robert Bradshaw (2007-09-19): added strip_string_literals, containing_block utility functions. Arrr!; added [1,2,..,n] notation

- Robert Bradshaw (2008-01-04): implicit multiplication (off by default)

- Robert Bradshaw (2008-09-23): factor out constants

- Robert Bradshaw (2009-01): simplify preparser by making it modular and using regular expressions; bug fixes, complex numbers, and binary input

class sage.repl.preparse.**QuoteStack**

> Bases: `object`
>
> The preserved state of parsing in *`strip_string_literals()`*.
>
> **peek**()
>
>> Get the frame at the top of the stack or None if empty.
>>
>> EXAMPLES:
>>
>> ```
>> sage: qs = sage.repl.preparse.QuoteStack()
>> sage: qs.peek()
>> sage: qs.push(sage.repl.preparse.QuoteStackFrame('"'))
>> sage: qs.peek()
>> QuoteStackFrame(...delim='"'...)
>> ```
>>
>> ```
>> >>> from sage.all import *
>> >>> qs = sage.repl.preparse.QuoteStack()
>> >>> qs.peek()
>> >>> qs.push(sage.repl.preparse.QuoteStackFrame('"'))
>> >>> qs.peek()
>> QuoteStackFrame(...delim='"'...)
>> ```
>
> **pop**()
>
>> Remove and return the frame that was most recently added to the stack.
>>
>> Raise an `IndexError` if the stack is empty.
>>
>> EXAMPLES:

```
sage: qs = sage.repl.preparse.QuoteStack()
sage: qs.pop()
Traceback (most recent call last):
...
IndexError: ...
sage: qs.push(sage.repl.preparse.QuoteStackFrame('"'))
sage: qs.pop()
QuoteStackFrame(...delim='"'...)
```

```
>>> from sage.all import *
>>> qs = sage.repl.preparse.QuoteStack()
>>> qs.pop()
Traceback (most recent call last):
...
IndexError: ...
>>> qs.push(sage.repl.preparse.QuoteStackFrame('"'))
>>> qs.pop()
QuoteStackFrame(...delim='"'...)
```

push(*frame*)

Add a frame to the stack.

If the frame corresponds to an F-string, its delimiter is marked as no longer being a *safe_delimiter()*.

EXAMPLES:

```
sage: qs = sage.repl.preparse.QuoteStack()
sage: qs.push(sage.repl.preparse.QuoteStackFrame("'"))
sage: len(qs)
1
```

```
>>> from sage.all import *
>>> qs = sage.repl.preparse.QuoteStack()
>>> qs.push(sage.repl.preparse.QuoteStackFrame("'"))
>>> len(qs)
1
```

safe_delimiter()

Return a string delimiter that may be safely inserted into the code output by *strip_string_literals()*, if any.

' is preferred over ". The triple-quoted versions are never returned since by the time they would be chosen, they would also be invalid. ''' cannot, for example, appear within an F-string delimited by '.

Once marked unsafe, a delimiter is never made safe again, even after the stack frame that used it is popped. It may no longer be applicable to parsing, but it appears somewhere in the processed code, so it is not safe to insert just anywhere. A future enhancement could be to map ranges in the processed code to the delimiter(s) that would be safe to insert there.

EXAMPLES:

```
sage: from sage.repl.preparse import QuoteStack, QuoteStackFrame
sage: s = QuoteStack()
sage: s.safe_delimiter()
```

```
"'"
sage: s.push(QuoteStackFrame("'"))
sage: s.safe_delimiter()
"'"
sage: s.pop()
QuoteStackFrame(...)
sage: s.push(QuoteStackFrame("'", f_string=True))
sage: s.safe_delimiter()
'"'
sage: s.push(QuoteStackFrame('"', f_string=True))
sage: s.safe_delimiter() is None
True
```

```
>>> from sage.all import *
>>> from sage.repl.preparse import QuoteStack, QuoteStackFrame
>>> s = QuoteStack()
>>> s.safe_delimiter()
"'"
>>> s.push(QuoteStackFrame("'"))
>>> s.safe_delimiter()
"'"
>>> s.pop()
QuoteStackFrame(...)
>>> s.push(QuoteStackFrame("'", f_string=True))
>>> s.safe_delimiter()
'"'
>>> s.push(QuoteStackFrame('"', f_string=True))
>>> s.safe_delimiter() is None
True
```

class sage.repl.preparse.**QuoteStackFrame**(*delim*, *raw=False*, *f_string=False*, *braces=0*, *parens=0*, *brackets=0*, *fmt_spec=False*, *nested_fmt_spec=False*)

Bases: SimpleNamespace

The state of a single level of a string literal being parsed.

Only F-strings have more than one level.

sage.repl.preparse.**containing_block**(*code*, *idx*, *delimiters=['()', '[]', '{}']*, *require_delim=True*)

Find the code block given by balanced delimiters that contains the position idx.

INPUT:

- code – string

- idx – integer; a starting position

- delimiters – list of strings (default: ['()', '[]', '{}']); the delimiters to balance. A delimiter must be a single character and no character can at the same time be opening and closing delimiter.

- require_delim – boolean (default: True); whether to raise a SyntaxError if delimiters are present. If the delimiters are unbalanced, an error will be raised in any case.

OUTPUT:

- a 2-tuple (a,b) of integers, such that code[a:b] is delimited by balanced delimiters, a<=idx<b, and a is maximal and b is minimal with that property. If that does not exist, a SyntaxError is raised.

- If `require_delim` is false and `a,b` as above can not be found, then `0, len(code)` is returned.

EXAMPLES:

```
sage: from sage.repl.preparse import containing_block
sage: s = "factor(next_prime(L[5]+1))"
sage: s[22]
'+'
sage: start, end = containing_block(s, 22)
sage: start, end
(17, 25)
sage: s[start:end]
'(L[5]+1)'
sage: s[20]
'5'
sage: start, end = containing_block(s, 20); s[start:end]
'[5]'
sage: start, end = containing_block(s, 20, delimiters=['()']); s[start:end]
'(L[5]+1)'
sage: start, end = containing_block(s, 10); s[start:end]
'(next_prime(L[5]+1))'
```

```
>>> from sage.all import *
>>> from sage.repl.preparse import containing_block
>>> s = "factor(next_prime(L[5]+1))"
>>> s[Integer(22)]
'+'
>>> start, end = containing_block(s, Integer(22))
>>> start, end
(17, 25)
>>> s[start:end]
'(L[5]+1)'
>>> s[Integer(20)]
'5'
>>> start, end = containing_block(s, Integer(20)); s[start:end]
'[5]'
>>> start, end = containing_block(s, Integer(20), delimiters=['()']); s[start:end]
'(L[5]+1)'
>>> start, end = containing_block(s, Integer(10)); s[start:end]
'(next_prime(L[5]+1))'
```

sage.repl.preparse.**extract_numeric_literals**(*code*)

Pulls out numeric literals and assigns them to global variables. This eliminates the need to re-parse and create the literals, e.g., during every iteration of a loop.

INPUT:

- `code` – string; a block of code

OUTPUT:

- a (string, string:string dictionary) 2-tuple; the block with literals replaced by variable names and a mapping from names to the new variables

EXAMPLES:

```
sage: from sage.repl.preparse import extract_numeric_literals
sage: code, nums = extract_numeric_literals("1.2 + 5")
sage: print(code)
_sage_const_1p2  + _sage_const_5
sage: print(nums)
{'_sage_const_1p2': "RealNumber('1.2')", '_sage_const_5': 'Integer(5)'}

sage: extract_numeric_literals("[1, 1.1, 1e1, -1e-1, 1.]")[0]
'[_sage_const_1 , _sage_const_1p1 , _sage_const_1e1 , -_sage_const_1en1 , _sage_
↪const_1p ]'

sage: extract_numeric_literals("[1.sqrt(), 1.2.sqrt(), 1r, 1.2r, R.1, R0.1, (1..
↪5)]")[0]
'[_sage_const_1 .sqrt(), _sage_const_1p2 .sqrt(), 1 , 1.2 , R.1, R0.1, (_sage_
↪const_1 .._sage_const_5 )]'
```

```
>>> from sage.all import *
>>> from sage.repl.preparse import extract_numeric_literals
>>> code, nums = extract_numeric_literals("1.2 + 5")
>>> print(code)
_sage_const_1p2  + _sage_const_5
>>> print(nums)
{'_sage_const_1p2': "RealNumber('1.2')", '_sage_const_5': 'Integer(5)'}

>>> extract_numeric_literals("[1, 1.1, 1e1, -1e-1, 1.]")[Integer(0)]
'[_sage_const_1 , _sage_const_1p1 , _sage_const_1e1 , -_sage_const_1en1 , _sage_
↪const_1p ]'

>>> extract_numeric_literals("[1.sqrt(), 1.2.sqrt(), 1r, 1.2r, R.1, R0.1, (1..5)]
↪")[Integer(0)]
'[_sage_const_1 .sqrt(), _sage_const_1p2 .sqrt(), 1 , 1.2 , R.1, R0.1, (_sage_
↪const_1 .._sage_const_5 )]'
```

sage.repl.preparse.**handle_encoding_declaration**(*contents*, *out*)

> Find a PEP 263-style Python encoding declaration in the first or second line of contents. If found, output it
> to out and return contents without the encoding line; otherwise output a default UTF-8 declaration and return
> contents.
>
> EXAMPLES:

```
sage: from sage.repl.preparse import handle_encoding_declaration
sage: import sys
sage: c1='# -*- coding: latin-1 -*-\nimport os, sys\n...'
sage: c2='# -*- coding: iso-8859-15 -*-\nimport os, sys\n...'
sage: c3='# -*- coding: ascii -*-\nimport os, sys\n...'
sage: c4='import os, sys\n...'
sage: handle_encoding_declaration(c1, sys.stdout)
# -*- coding: latin-1 -*-
'import os, sys\n...'
sage: handle_encoding_declaration(c2, sys.stdout)
# -*- coding: iso-8859-15 -*-
'import os, sys\n...'
sage: handle_encoding_declaration(c3, sys.stdout)
```

(continues on next page)

```
# -*- coding: ascii -*-
'import os, sys\n...'
sage: handle_encoding_declaration(c4, sys.stdout)
# -*- coding: utf-8 -*-
'import os, sys\n...'
```

```
>>> from sage.all import *
>>> from sage.repl.preparse import handle_encoding_declaration
>>> import sys
>>> c1='# -*- coding: latin-1 -*-\nimport os, sys\n...'
>>> c2='# -*- coding: iso-8859-15 -*-\nimport os, sys\n...'
>>> c3='# -*- coding: ascii -*-\nimport os, sys\n...'
>>> c4='import os, sys\n...'
>>> handle_encoding_declaration(c1, sys.stdout)
# -*- coding: latin-1 -*-
'import os, sys\n...'
>>> handle_encoding_declaration(c2, sys.stdout)
# -*- coding: iso-8859-15 -*-
'import os, sys\n...'
>>> handle_encoding_declaration(c3, sys.stdout)
# -*- coding: ascii -*-
'import os, sys\n...'
>>> handle_encoding_declaration(c4, sys.stdout)
# -*- coding: utf-8 -*-
'import os, sys\n...'
```

> **ⓘ Note**
>
> - **PEP 263** says that Python will interpret a UTF-8 byte order mark as a declaration of UTF-8 encoding, but I do not think we do that; this function only sees a Python string so it cannot account for a BOM.
>
> - We default to UTF-8 encoding even though PEP 263 says that Python files should default to ASCII.
>
> - Also see https://docs.python.org/ref/encodings.html.

AUTHORS:

- Lars Fischer

- Dan Drake (2010-12-08, rewrite for Issue #10440)

sage.repl.preparse.**implicit_mul**(*code*, *level=5*)

Insert *'s to make implicit multiplication explicit.

INPUT:

- `code` – string; the code with missing *'s

- `level` – integer (default: 5); see *implicit_multiplication()* for a list

OUTPUT: string

EXAMPLES:

```
sage: from sage.repl.preparse import implicit_mul
sage: implicit_mul('(2x^2-4x+3)a0')
'(2*x^2-4*x+3)*a0'
sage: implicit_mul('a b c in L')
'a*b*c in L'
sage: implicit_mul('1r + 1e3 + 5exp(2)')
'1r + 1e3 + 5*exp(2)'
sage: implicit_mul('f(a)(b)', level=10)
'f(a)*(b)'
```

```
>>> from sage.all import *
>>> from sage.repl.preparse import implicit_mul
>>> implicit_mul('(2x^2-4x+3)a0')
'(2*x^2-4*x+3)*a0'
>>> implicit_mul('a b c in L')
'a*b*c in L'
>>> implicit_mul('1r + 1e3 + 5exp(2)')
'1r + 1e3 + 5*exp(2)'
>>> implicit_mul('f(a)(b)', level=Integer(10))
'f(a)*(b)'
```

sage.repl.preparse.**implicit_multiplication**(*level=None*)

> Turn implicit multiplication on or off, optionally setting a specific `level`.
>
> INPUT:
>
> > - `level` – boolean or integer (default: 5); how aggressive to be in placing *'s
> >
> > > - 0 – Do nothing
> > >
> > > - 1 – Numeric followed by alphanumeric
> > >
> > > - 2 – Closing parentheses followed by alphanumeric
> > >
> > > - 3 – Spaces between alphanumeric
> > >
> > > - 10 – Adjacent parentheses (may mangle call statements)
>
> OUTPUT: the current `level` if no argument is given
>
> EXAMPLES:

```
sage: implicit_multiplication(True)
sage: implicit_multiplication()
5
sage: preparse('2x')
'Integer(2)*x'
sage: implicit_multiplication(False)
sage: preparse('2x')
'2x'
```

```
>>> from sage.all import *
>>> implicit_multiplication(True)
>>> implicit_multiplication()
5
>>> preparse('2x')
```

```
'Integer(2)*x'
>>> implicit_multiplication(False)
>>> preparse('2x')
'2x'
```

Note that the IPython automagic feature cannot be used if `level >= 3`:

```
sage: implicit_multiplication(3)
sage: preparse('cd Documents')
'cd*Documents'
sage: implicit_multiplication(2)
sage: preparse('cd Documents')
'cd Documents'
sage: implicit_multiplication(False)
```

```
>>> from sage.all import *
>>> implicit_multiplication(Integer(3))
>>> preparse('cd Documents')
'cd*Documents'
>>> implicit_multiplication(Integer(2))
>>> preparse('cd Documents')
'cd Documents'
>>> implicit_multiplication(False)
```

In this case, one can use the explicit syntax for IPython magics such as `%cd Documents`.

sage.repl.preparse.**in_quote**()

sage.repl.preparse.**isalphadigit_**(*s*)

　　Return `True` if s is a non-empty string of alphabetic characters or a non-empty string of digits or just a single _

　　EXAMPLES:

```
sage: from sage.repl.preparse import isalphadigit_
sage: isalphadigit_('abc')
True
sage: isalphadigit_('123')
True
sage: isalphadigit_('_')
True
sage: isalphadigit_('a123')
False
```

```
>>> from sage.all import *
>>> from sage.repl.preparse import isalphadigit_
>>> isalphadigit_('abc')
True
>>> isalphadigit_('123')
True
>>> isalphadigit_('_')
True
>>> isalphadigit_('a123')
False
```

sage.repl.preparse.**parse_ellipsis**(*code*, *preparse_step=True*)

> Preparses [0,2,..,n] notation.
>
> INPUT:
>
> > • `code` – string
> >
> > • `preparse_step` – boolean (default: `True`)
>
> OUTPUT: string
>
> EXAMPLES:

```
sage: from sage.repl.preparse import parse_ellipsis
sage: parse_ellipsis("[1,2,..,n]")
'(ellipsis_range(1,2,Ellipsis,n))'
sage: parse_ellipsis("for i in (f(x) .. L[10]):")
'for i in (ellipsis_iter(f(x) ,Ellipsis, L[10])):'
sage: [1.0..2.0]                                                      #␣
↪needs sage.rings.real_mpfr
[1.00000000000000, 2.00000000000000]
```

```
>>> from sage.all import *
>>> from sage.repl.preparse import parse_ellipsis
>>> parse_ellipsis("[1,2,..,n]")
'(ellipsis_range(1,2,Ellipsis,n))'
>>> parse_ellipsis("for i in (f(x) .. L[10]):")
'for i in (ellipsis_iter(f(x) ,Ellipsis, L[10])):'
>>> (ellipsis_range(RealNumber('1.0'),Ellipsis,RealNumber('2.0')))          ␣
↪                                              # needs sage.rings.real_mpfr
[1.00000000000000, 2.00000000000000]
```

sage.repl.preparse.**preparse**(*line*, *reset=True*, *do_time=False*, *ignore_prompts=False*, *numeric_literals=True*)

> Preparse a line of input.
>
> INPUT:
>
> > • `line` – string
> >
> > • `reset` – boolean (default: `True`)
> >
> > • `do_time` – boolean (default: `False`)
> >
> > • `ignore_prompts` – boolean (default: `False`)
> >
> > • `numeric_literals` – boolean (default: `True`)
>
> OUTPUT: string
>
> EXAMPLES:

```
sage: preparse("ZZ.<x> = ZZ['x']")
"ZZ = ZZ['x']; (x,) = ZZ._first_ngens(1)"
sage: preparse("ZZ.<x> = ZZ['y']")
"ZZ = ZZ['y']; (x,) = ZZ._first_ngens(1)"
sage: preparse("ZZ.<x,y> = ZZ[]")
"ZZ = ZZ['x, y']; (x, y,) = ZZ._first_ngens(2)"
sage: preparse("ZZ.<x,y> = ZZ['u,v']")
"ZZ = ZZ['u,v']; (x, y,) = ZZ._first_ngens(2)"
```

(continues on next page)

```
sage: preparse("ZZ.<x> = QQ[2^(1/3)]")
'ZZ = QQ[Integer(2)**(Integer(1)/Integer(3))]; (x,) = ZZ._first_ngens(1)'
sage: QQ[2^(1/3)]                                                          #␣
→needs sage.rings.number_field sage.symbolic
Number Field in a with defining polynomial x^3 - 2 with a = 1.259921049894873?

sage: preparse("a^b")
'a**b'
sage: preparse("a^^b")
'a^b'
sage: 8^1
8
sage: 8^^1
9
sage: 9^^1
8

sage: preparse("A \\ B")
'A  * BackslashOperator() * B'
sage: preparse("A^2 \\ B + C")
'A**Integer(2)  * BackslashOperator() * B + C'
sage: preparse("a \\ b \\\\") # There is really only one backslash here, it is just␣
→being escaped.
'a  * BackslashOperator() * b \\'

sage: preparse("time R.<x> = ZZ[]", do_time=True)
'__time__ = cputime(); __wall__ = walltime(); R = ZZ[\'x\']; print("Time: CPU {:.
→2f} s, Wall: {:.2f} s".format(cputime(__time__), walltime(__wall__))); (x,) = R.
→_first_ngens(1)'
```

```
>>> from sage.all import *
>>> preparse("ZZ.<x> = ZZ['x']")
"ZZ = ZZ['x']; (x,) = ZZ._first_ngens(1)"
>>> preparse("ZZ.<x> = ZZ['y']")
"ZZ = ZZ['y']; (x,) = ZZ._first_ngens(1)"
>>> preparse("ZZ.<x,y> = ZZ[]")
"ZZ = ZZ['x, y']; (x, y,) = ZZ._first_ngens(2)"
>>> preparse("ZZ.<x,y> = ZZ['u,v']")
"ZZ = ZZ['u,v']; (x, y,) = ZZ._first_ngens(2)"
>>> preparse("ZZ.<x> = QQ[2^(1/3)]")
'ZZ = QQ[Integer(2)**(Integer(1)/Integer(3))]; (x,) = ZZ._first_ngens(1)'
>>> QQ[Integer(2)**(Integer(1)/Integer(3))]                                  ␣
→                            # needs sage.rings.number_field sage.symbolic
Number Field in a with defining polynomial x^3 - 2 with a = 1.259921049894873?

>>> preparse("a^b")
'a**b'
>>> preparse("a^^b")
'a^b'
>>> Integer(8)**Integer(1)
8
>>> Integer(8)^Integer(1)
```

```
9
>>> Integer(9)^Integer(1)
8

>>> preparse("A \\ B")
'A  * BackslashOperator() * B'
>>> preparse("A^2 \\ B + C")
'A**Integer(2)  * BackslashOperator() * B + C'
>>> preparse("a \\ b \\") # There is really only one backslash here, it is just␣
→being escaped.
'a  * BackslashOperator() * b \\'

>>> preparse("time R.<x> = ZZ[]", do_time=True)
'__time__ = cputime(); __wall__ = walltime(); R = ZZ[\'x\']; print("Time: CPU {:.
→2f} s, Wall: {:.2f} s".format(cputime(__time__), walltime(__wall__))); (x,) = R.
→_first_ngens(1)'
```

sage.repl.preparse.**preparse_calculus**(*code*)

Supports calculus-like function assignment, e.g., transforms:

```
f(x,y,z) = sin(x^3 - 4*y) + y^x
```

into:

```
__tmp__=var("x,y,z")
f = symbolic_expression(sin(x**3 - 4*y) + y**x).function(x,y,z)
```

AUTHORS:

- Bobby Moretti

    - Initial version - 02/2007

- William Stein

    - Make variables become defined if they are not already defined.

- Robert Bradshaw

    - Rewrite using regular expressions (01/2009)

EXAMPLES:

```
sage: preparse("f(x) = x^3-x")
'__tmp__=var("x"); f = symbolic_expression(x**Integer(3)-x).function(x)'
sage: preparse("f(u,v) = u - v")
'__tmp__=var("u,v"); f = symbolic_expression(u - v).function(u,v)'
sage: preparse("f(x) =-5")
'__tmp__=var("x"); f = symbolic_expression(-Integer(5)).function(x)'
sage: preparse("f(x) -= 5")
'f(x) -= Integer(5)'
sage: preparse("f(x_1, x_2) = x_1^2 - x_2^2")
'__tmp__=var("x_1,x_2"); f = symbolic_expression(x_1**Integer(2) - x_
→2**Integer(2)).function(x_1,x_2)'
```

```
>>> from sage.all import *
>>> preparse("f(x) = x^3-x")
'__tmp__=var("x"); f = symbolic_expression(x**Integer(3)-x).function(x)'
>>> preparse("f(u,v) = u - v")
'__tmp__=var("u,v"); f = symbolic_expression(u - v).function(u,v)'
>>> preparse("f(x) =-5")
'__tmp__=var("x"); f = symbolic_expression(-Integer(5)).function(x)'
>>> preparse("f(x) -= 5")
'f(x) -= Integer(5)'
>>> preparse("f(x_1, x_2) = x_1^2 - x_2^2")
'__tmp__=var("x_1,x_2"); f = symbolic_expression(x_1**Integer(2) - x_
↪2**Integer(2)).function(x_1,x_2)'
```

For simplicity, this function assumes all statements begin and end with a semicolon:

```
sage: from sage.repl.preparse import preparse_calculus
sage: preparse_calculus(";f(t,s)=t^2;")
';__tmp__=var("t,s"); f = symbolic_expression(t^2).function(t,s);'
sage: preparse_calculus(";f( t , s ) = t^2;")
';__tmp__=var("t,s"); f = symbolic_expression(t^2).function(t,s);'
```

```
>>> from sage.all import *
>>> from sage.repl.preparse import preparse_calculus
>>> preparse_calculus(";f(t,s)=t^2;")
';__tmp__=var("t,s"); f = symbolic_expression(t^2).function(t,s);'
>>> preparse_calculus(";f( t , s ) = t^2;")
';__tmp__=var("t,s"); f = symbolic_expression(t^2).function(t,s);'
```

sage.repl.preparse.**preparse_file**(*contents*, *globals=None*, *numeric_literals=True*)

Preparse `contents` which is input from a file such as `.sage` files.

Special attentions are given to numeric literals and load/attach file directives.

> **ⓘ Note**
>
> Temporarily, if @parallel is in the input, then numeric_literals is always set to False.

INPUT:

- `contents` – string

- `globals` – dictionary or `None` (default: `None`); if given, then arguments to load/attach are evaluated in the namespace of this dictionary

- `numeric_literals` – boolean (default: `True`); whether to factor out wrapping of integers and floats, so they do not get created repeatedly inside loops

OUTPUT: string

sage.repl.preparse.**preparse_file_named**(*name*)

Preparse file named `name` (presumably a `.sage` file), outputting to a temporary file.

This returns the temporary file as a `Path` object.

EXAMPLES:

```
sage: from sage.repl.preparse import preparse_file_named
sage: tmpf = tmp_filename(ext='.sage')
sage: with open(tmpf, 'w') as f:
....:     out = f.write("a = 2")
sage: preparse_file_named(tmpf)
PosixPath('...sage.py')
```

```
>>> from sage.all import *
>>> from sage.repl.preparse import preparse_file_named
>>> tmpf = tmp_filename(ext='.sage')
>>> with open(tmpf, 'w') as f:
...     out = f.write("a = 2")
>>> preparse_file_named(tmpf)
PosixPath('...sage.py')
```

sage.repl.preparse.**preparse_file_named_to_stream**(*name*, *out*)

> Preparse file named code{name} (presumably a .sage file), outputting to stream code{out}.

sage.repl.preparse.**preparse_generators**(*code*)

> Parse generator syntax, converting:

```
obj.<gen0,gen1,...,genN> = objConstructor(...)
```

> into:

```
obj = objConstructor(..., names=("gen0", "gen1", ..., "genN"))
(gen0, gen1, ..., genN,) = obj.gens()
```

> and:

```
obj.<gen0,gen1,...,genN> = R[interior]
```

> into:

```
obj = R[interior]; (gen0, gen1, ..., genN,) = obj.gens()
```

> INPUT:
>
> * code – string
>
> OUTPUT: string
>
> LIMITATIONS:
>
> * The entire constructor *must* be on one line.
>
> AUTHORS:
>
> * 2006-04-14: Joe Wetherell (jlwether@alum.mit.edu)
>
>   – Initial version.
>
> * 2006-04-17: William Stein
>
>   – Improvements to allow multiple statements.
>
> * 2006-05-01: William
>
>   – Fix bug that Joe found.

- 2006-10-31: William

    - Fix so obj does not have to be mutated.

- 2009-01-27: Robert Bradshaw

    - Rewrite using regular expressions

`sage.repl.preparse.`**`preparse_numeric_literals`**(*code*, *extract=False*, *quotes="'"*)

Preparse numerical literals into their Sage counterparts, e.g. Integer, RealNumber, and ComplexNumber.

INPUT:

- `code` – string; a code block to preparse

- `extract` – boolean (default: `False`); whether to create names for the literals and return a dictionary of name-construction pairs

- `quotes` – string (default: `"'"`); used to surround string arguments to RealNumber and ComplexNumber. If `None`, will rebuild the string using a list of its Unicode code-points.

OUTPUT:

- a string or (string, string:string dictionary) 2-tuple; the preparsed block and, if `extract` is True, the name-construction mapping

EXAMPLES:

```
sage: from sage.repl.preparse import preparse_numeric_literals
sage: preparse_numeric_literals("5")
'Integer(5)'
sage: preparse_numeric_literals("5j")
"ComplexNumber(0, '5')"
sage: preparse_numeric_literals("5jr")
'5J'
sage: preparse_numeric_literals("5l")
'5'
sage: preparse_numeric_literals("5L")
'5'
sage: preparse_numeric_literals("1.5")
"RealNumber('1.5')"
sage: preparse_numeric_literals("1.5j")
"ComplexNumber(0, '1.5')"
sage: preparse_numeric_literals(".5j")
"ComplexNumber(0, '.5')"
sage: preparse_numeric_literals("5e9j")
"ComplexNumber(0, '5e9')"
sage: preparse_numeric_literals("5.")
"RealNumber('5.')"
sage: preparse_numeric_literals("5.j")
"ComplexNumber(0, '5.')"
sage: preparse_numeric_literals("5.foo()")
'Integer(5).foo()'
sage: preparse_numeric_literals("5.5.foo()")
"RealNumber('5.5').foo()"
sage: preparse_numeric_literals("5.5j.foo()")
"ComplexNumber(0, '5.5').foo()"
sage: preparse_numeric_literals("5j.foo()")
"ComplexNumber(0, '5').foo()"
```

(continues on next page)

```
sage: preparse_numeric_literals("1.exp()")
'Integer(1).exp()'
sage: preparse_numeric_literals("1e+10")
"RealNumber('1e+10')"
sage: preparse_numeric_literals("0x0af")
'Integer(0x0af)'
sage: preparse_numeric_literals("0x10.sqrt()")
'Integer(0x10).sqrt()'
sage: preparse_numeric_literals('0o100')
'Integer(0o100)'
sage: preparse_numeric_literals('0b111001')
'Integer(0b111001)'
sage: preparse_numeric_literals('0xe')
'Integer(0xe)'
sage: preparse_numeric_literals('0xEAR')
'0xEA'
sage: preparse_numeric_literals('0x1012Fae')
'Integer(0x1012Fae)'
sage: preparse_numeric_literals('042')
'Integer(42)'
sage: preparse_numeric_literals('000042')
'Integer(42)'
```

```
>>> from sage.all import *
>>> from sage.repl.preparse import preparse_numeric_literals
>>> preparse_numeric_literals("5")
'Integer(5)'
>>> preparse_numeric_literals("5j")
"ComplexNumber(0, '5')"
>>> preparse_numeric_literals("5jr")
'5J'
>>> preparse_numeric_literals("5l")
'5'
>>> preparse_numeric_literals("5L")
'5'
>>> preparse_numeric_literals("1.5")
"RealNumber('1.5')"
>>> preparse_numeric_literals("1.5j")
"ComplexNumber(0, '1.5')"
>>> preparse_numeric_literals(".5j")
"ComplexNumber(0, '.5')"
>>> preparse_numeric_literals("5e9j")
"ComplexNumber(0, '5e9')"
>>> preparse_numeric_literals("5.")
"RealNumber('5.')"
>>> preparse_numeric_literals("5.j")
"ComplexNumber(0, '5.')"
>>> preparse_numeric_literals("5.foo()")
'Integer(5).foo()'
>>> preparse_numeric_literals("5.5.foo()")
"RealNumber('5.5').foo()"
>>> preparse_numeric_literals("5.5j.foo()")
```

```
"ComplexNumber(0, '5.5').foo()"
>>> preparse_numeric_literals("5j.foo()")
"ComplexNumber(0, '5').foo()"
>>> preparse_numeric_literals("1.exp()")
'Integer(1).exp()'
>>> preparse_numeric_literals("1e+10")
"RealNumber('1e+10')"
>>> preparse_numeric_literals("0x0af")
'Integer(0x0af)'
>>> preparse_numeric_literals("0x10.sqrt()")
'Integer(0x10).sqrt()'
>>> preparse_numeric_literals('0o100')
'Integer(0o100)'
>>> preparse_numeric_literals('0b111001')
'Integer(0b111001)'
>>> preparse_numeric_literals('0xe')
'Integer(0xe)'
>>> preparse_numeric_literals('0xEAR')
'0xEA'
>>> preparse_numeric_literals('0x1012Fae')
'Integer(0x1012Fae)'
>>> preparse_numeric_literals('042')
'Integer(42)'
>>> preparse_numeric_literals('000042')
'Integer(42)'
```

Test underscores as digit separators (PEP 515, https://www.python.org/dev/peps/pep-0515/):

```
sage: preparse_numeric_literals('123_456')
'Integer(123_456)'
sage: preparse_numeric_literals('123_456.78_9_0')
"RealNumber('123_456.78_9_0')"
sage: preparse_numeric_literals('0b11_011')
'Integer(0b11_011)'
sage: preparse_numeric_literals('0o76_321')
'Integer(0o76_321)'
sage: preparse_numeric_literals('0xaa_aaa')
'Integer(0xaa_aaa)'
sage: preparse_numeric_literals('1_3.2_5e-2_2')
"RealNumber('1_3.2_5e-2_2')"

sage: for f in ["1_1.", "11_2.", "1.1_1", "1_1.1_1", ".1_1", ".1_1e1_1", ".1e1_1",
....:           "1e12_3", "1_1e1_1", "1.1_3e1_2", "1_1e1_1", "1e1", "1.e1_1",
....:           "1.0", "1_1.0"]:
....:     preparse_numeric_literals(f)
....:     assert preparse(f) == preparse_numeric_literals(f), f
"RealNumber('1_1.')"
"RealNumber('11_2.')"
"RealNumber('1.1_1')"
"RealNumber('1_1.1_1')"
"RealNumber('.1_1')"
"RealNumber('.1_1e1_1')"
```

```
"RealNumber('.1e1_1')"
"RealNumber('1e12_3')"
"RealNumber('1_1e1_1')"
"RealNumber('1.1_3e1_2')"
"RealNumber('1_1e1_1')"
"RealNumber('1e1')"
"RealNumber('1.e1_1')"
"RealNumber('1.0')"
"RealNumber('1_1.0')"
```

```
>>> from sage.all import *
>>> preparse_numeric_literals('123_456')
'Integer(123_456)'
>>> preparse_numeric_literals('123_456.78_9_0')
"RealNumber('123_456.78_9_0')"
>>> preparse_numeric_literals('0b11_011')
'Integer(0b11_011)'
>>> preparse_numeric_literals('0o76_321')
'Integer(0o76_321)'
>>> preparse_numeric_literals('0xaa_aaa')
'Integer(0xaa_aaa)'
>>> preparse_numeric_literals('1_3.2_5e-2_2')
"RealNumber('1_3.2_5e-2_2')"

>>> for f in ["1_1.", "11_2.", "1.1_1", "1_1.1_1", ".1_1", ".1_1e1_1", ".1e1_1",
...           "1e12_3", "1_1e1_1", "1.1_3e1_2", "1_1e1_1", "1e1", "1.e1_1",
...           "1.0", "1_1.0"]:
...     preparse_numeric_literals(f)
...     assert preparse(f) == preparse_numeric_literals(f), f
"RealNumber('1_1.')"
"RealNumber('11_2.')"
"RealNumber('1.1_1')"
"RealNumber('1_1.1_1')"
"RealNumber('.1_1')"
"RealNumber('.1_1e1_1')"
"RealNumber('.1e1_1')"
"RealNumber('1e12_3')"
"RealNumber('1_1e1_1')"
"RealNumber('1.1_3e1_2')"
"RealNumber('1_1e1_1')"
"RealNumber('1e1')"
"RealNumber('1.e1_1')"
"RealNumber('1.0')"
"RealNumber('1_1.0')"
```

Having consecutive underscores is not valid Python syntax, so it is not preparsed, and similarly with a trailing underscore:

```
sage: preparse_numeric_literals('123__45')
'123__45'
sage: 123__45
Traceback (most recent call last):
```

```
...
SyntaxError: invalid ...

sage: preparse_numeric_literals('3040_1_')
'3040_1_'
sage: 3040_1_
Traceback (most recent call last):
...
SyntaxError: invalid ...
```

```
>>> from sage.all import *
>>> preparse_numeric_literals('123__45')
'123__45'
>>> 123__45
Traceback (most recent call last):
...
SyntaxError: invalid ...

>>> preparse_numeric_literals('3040_1_')
'3040_1_'
>>> 3040_1_
Traceback (most recent call last):
...
SyntaxError: invalid ...
```

Using the `quotes` parameter:

```
sage: preparse_numeric_literals('5j', quotes='"')
'ComplexNumber(0, "5")'
sage: preparse_numeric_literals('3.14', quotes="'''")
"RealNumber('''3.14''')"
sage: preparse_numeric_literals('3.14', quotes=None)
'RealNumber(str().join(map(chr, [51, 46, 49, 52])))'
sage: preparse_numeric_literals('5j', quotes=None)
'ComplexNumber(0, str().join(map(chr, [53])))'
```

```
>>> from sage.all import *
>>> preparse_numeric_literals('5j', quotes='"')
'ComplexNumber(0, "5")'
>>> preparse_numeric_literals('3.14', quotes="'''")
"RealNumber('''3.14''')"
>>> preparse_numeric_literals('3.14', quotes=None)
'RealNumber(str().join(map(chr, [51, 46, 49, 52])))'
>>> preparse_numeric_literals('5j', quotes=None)
'ComplexNumber(0, str().join(map(chr, [53])))'
```

sage.repl.preparse.**strip_prompts**(*line*)

> Remove leading sage: and >>> prompts so that pasting of examples from the documentation works.
>
> INPUT:
>
> > • `line` – string to process
>
> OUTPUT: string stripped of leading prompts

EXAMPLES:

```
sage: from sage.repl.preparse import strip_prompts
sage: strip_prompts("sage: 2 + 2")
'2 + 2'
sage: strip_prompts(">>>   3 + 2")
'3 + 2'
sage: strip_prompts("  2 + 4")
'  2 + 4'
```

```
>>> from sage.all import *
>>> from sage.repl.preparse import strip_prompts
>>> strip_prompts("sage: 2 + 2")
'2 + 2'
>>> strip_prompts(">>>   3 + 2")
'3 + 2'
>>> strip_prompts("  2 + 4")
'  2 + 4'
```

sage.repl.preparse.**strip_string_literals**(*code*, *state=None*)

Return a string with all literal quotes replaced with labels and a dictionary of labels for re-substitution.

This makes parsing easier.

INPUT:

- code – string; the input

- state – a *QuoteStack* (default: None); state with which to continue processing, e.g., across multiple calls to this function

OUTPUT:

- a 3-tuple of the processed code, the dictionary of labels, and any accumulated state

EXAMPLES:

```
sage: from sage.repl.preparse import strip_string_literals
sage: s, literals, state = strip_string_literals(r'''['a', "b", 'c', "d\""]''')
sage: s
'[%(L1)s, %(L2)s, %(L3)s, %(L4)s]'
sage: literals
{'L1': "'a'", 'L2': '"b"', 'L3': "'c'", 'L4': '"d\\""'}
sage: print(s % literals)
['a', "b", 'c', "d\""]
sage: print(strip_string_literals(r'-"\\""-"\\"-')[0])
-%(L1)s-%(L2)s-
```

```
>>> from sage.all import *
>>> from sage.repl.preparse import strip_string_literals
>>> s, literals, state = strip_string_literals(r'''['a', "b", 'c', "d\""]''')
>>> s
'[%(L1)s, %(L2)s, %(L3)s, %(L4)s]'
>>> literals
{'L1': "'a'", 'L2': '"b"', 'L3': "'c'", 'L4': '"d\\""'}
>>> print(s % literals)
```

```
['a', "b", 'c', "d\""]
>>> print(strip_string_literals(r'-"\\""-"\\"-')[Integer(0)])
-%(L1)s-%(L2)s-
```

Triple-quotes are handled as well:

```
sage: s, literals, state = strip_string_literals("[a, '''b''', c, '']")
sage: s
'[a, %(L1)s, c, %(L2)s]'
sage: print(s % literals)
[a, '''b''', c, '']
```

```
>>> from sage.all import *
>>> s, literals, state = strip_string_literals("[a, '''b''', c, '']")
>>> s
'[a, %(L1)s, c, %(L2)s]'
>>> print(s % literals)
[a, '''b''', c, '']
```

Comments are substitute too:

```
sage: s, literals, state = strip_string_literals("code '#' # ccc 't'"); s
'code %(L1)s #%(L2)s'
sage: s % literals
"code '#' # ccc 't'"
```

```
>>> from sage.all import *
>>> s, literals, state = strip_string_literals("code '#' # ccc 't'"); s
'code %(L1)s #%(L2)s'
>>> s % literals
"code '#' # ccc 't'"
```

A state is returned so one can break strings across multiple calls to this function:

```
sage: s, literals, state = strip_string_literals('s = "some'); s
's = %(L1)s'
sage: s, literals, state = strip_string_literals('thing" * 5', state); s
'%(L1)s * 5'
```

```
>>> from sage.all import *
>>> s, literals, state = strip_string_literals('s = "some'); s
's = %(L1)s'
>>> s, literals, state = strip_string_literals('thing" * 5', state); s
'%(L1)s * 5'
```

LOADING AND ATTACHING FILES

Sage or Python files can be loaded (similar to Python's `execfile`) in a Sage session. Attaching is similar, except that the attached file is reloaded whenever it is changed.

3.1 Load Python, Sage, Cython, Fortran and Magma files in Sage

sage.repl.load.**is_loadable_filename**(*filename*)

> Return whether a file can be loaded into Sage.
>
> This checks only whether its name ends in one of the supported extensions `.py`, `.pyx`, `.sage`, `.spyx`, `.f`, `.f90` and `.m`.
>
> > **ⓘ Note**
> >
> > *load()* assumes that *.m* signifies a Magma file.
>
> INPUT:
>
> > • `filename` – string or `Path` object
>
> OUTPUT: boolean
>
> EXAMPLES:

```
sage: sage.repl.load.is_loadable_filename('foo.bar')
False
sage: sage.repl.load.is_loadable_filename('foo.c')
False
sage: sage.repl.load.is_loadable_filename('foo.sage')
True
sage: sage.repl.load.is_loadable_filename('FOO.F90')
True
sage: sage.repl.load.is_loadable_filename('foo.m')
True

sage: from pathlib import Path
sage: sage.repl.load.is_loadable_filename(Path('foo.py'))
True
```

```
>>> from sage.all import *
>>> sage.repl.load.is_loadable_filename('foo.bar')
```

```
False
>>> sage.repl.load.is_loadable_filename('foo.c')
False
>>> sage.repl.load.is_loadable_filename('foo.sage')
True
>>> sage.repl.load.is_loadable_filename('FOO.F90')
True
>>> sage.repl.load.is_loadable_filename('foo.m')
True

>>> from pathlib import Path
>>> sage.repl.load.is_loadable_filename(Path('foo.py'))
True
```

sage.repl.load.**load**(*filename*, *globals*, *attach=False*)

> Execute a file in the scope given by `globals`. If the name starts with `http://` or `https://`, it is treated as a URL and downloaded.

> > **ⓘ Note**
> >
> > For Cython files, the situation is more complicated – the module is first compiled to a temporary module `t` and executed via:
> >
> > ```
> > from t import *
> > ```

> > **ⓘ Note**
> >
> > The global `load` function is `sage.misc.persist.load()`, which delegates to this function for code file formats.
> >
> > `%runfile` magic can also be used, see *runfile()*.

> INPUT:
>
> - `filename` – string (denoting a filename or URL) or a `Path` object
>
> - `globals` – string:object dictionary; the context in which to execute the file contents
>
> - `attach` – boolean (default: `False`); whether to add the file to the list of attached files

> Loading an executable Sage script from the *command line* will run whatever code is inside an

> ```
> if __name__ == "__main__":
> ```

> section, as the condition on `__name__` will hold true (code run from the command line is considered to be running in the `__main__` module.)

> EXAMPLES:

> Note that `.py` files are *not* preparsed:

> ```
> sage: t = tmp_filename(ext='.py')
> sage: with open(t, 'w') as f:
> ```

```
....:     _ = f.write("print(('hi', 2^3)); z = -2^7")
sage: z = 1
sage: sage.repl.load.load(t, globals())
('hi', 1)
sage: z
-7
```

```
>>> from sage.all import *
>>> t = tmp_filename(ext='.py')
>>> with open(t, 'w') as f:
...     _ = f.write("print(('hi', 2^3)); z = -2^7")
>>> z = Integer(1)
>>> sage.repl.load.load(t, globals())
('hi', 1)
>>> z
-7
```

A `.sage` file *is* preparsed:

```
sage: t = tmp_filename(ext='.sage')
sage: with open(t, 'w') as f:
....:     _ = f.write("print(('hi', 2^3)); z = -2^7")
sage: z = 1
sage: sage.repl.load.load(t, globals())
('hi', 8)
sage: z
-128
```

```
>>> from sage.all import *
>>> t = tmp_filename(ext='.sage')
>>> with open(t, 'w') as f:
...     _ = f.write("print(('hi', 2^3)); z = -2^7")
>>> z = Integer(1)
>>> sage.repl.load.load(t, globals())
('hi', 8)
>>> z
-128
```

Cython files are *not* preparsed:

```
sage: t = tmp_filename(ext='.pyx')
sage: with open(t, 'w') as f:
....:     _ = f.write("print(('hi', 2^3)); z = -2^7")
sage: z = 1
sage: sage.repl.load.load(t, globals())                                        #␣
↪needs sage.misc.cython
Compiling ...
('hi', 1)
sage: z
-7
```

```
>>> from sage.all import *
>>> t = tmp_filename(ext='.pyx')
>>> with open(t, 'w') as f:
...     _ = f.write("print(('hi', 2^3)); z = -2^7")
>>> z = Integer(1)
>>> sage.repl.load.load(t, globals())                                    #␣
→needs sage.misc.cython
Compiling ...
('hi', 1)
>>> z
-7
```

If the file is not a Cython, Python, or Sage file, a `ValueError` is raised:

```
sage: sage.repl.load.load(tmp_filename(ext='.foo'), globals())
Traceback (most recent call last):
...
ValueError: unknown file extension '.foo' for load or attach (supported␣
→extensions: .py, .pyx, .sage, .spyx, .f, .f90, .m)
```

```
>>> from sage.all import *
>>> sage.repl.load.load(tmp_filename(ext='.foo'), globals())
Traceback (most recent call last):
...
ValueError: unknown file extension '.foo' for load or attach (supported␣
→extensions: .py, .pyx, .sage, .spyx, .f, .f90, .m)
```

We load a file given at a remote URL (not tested for security reasons):

```
sage: sage.repl.load.load('https://www.sagemath.org/files/loadtest.py',␣
→globals())  # not tested
hi from the net
5
```

```
>>> from sage.all import *
>>> sage.repl.load.load('https://www.sagemath.org/files/loadtest.py', globals())
→# not tested
hi from the net
5
```

We can load files using secure http (https):

```
sage: sage.repl.load.load('https://raw.githubusercontent.com/sagemath/sage-
→patchbot/3.0.0/sage_patchbot/util.py', globals())  # optional - internet
```

```
>>> from sage.all import *
>>> sage.repl.load.load('https://raw.githubusercontent.com/sagemath/sage-patchbot/
→3.0.0/sage_patchbot/util.py', globals())  # optional - internet
```

We attach a file (note that `attach()` is equivalent, but available at the global scope by default):

```
sage: t = tmp_filename(ext='.py')
sage: with open(t, 'w') as f:
```

(continues on next page)

```
....:     _ = f.write("print('hello world')")
sage: sage.repl.load.load(t, globals(), attach=True)
hello world
sage: t in attached_files()
True
```

```
>>> from sage.all import *
>>> t = tmp_filename(ext='.py')
>>> with open(t, 'w') as f:
...     _ = f.write("print('hello world')")
>>> sage.repl.load.load(t, globals(), attach=True)
hello world
>>> t in attached_files()
True
```

You cannot attach remote URLs (yet):

```
sage: sage.repl.load.load('https://www.sagemath.org/files/loadtest.py', globals(),
↪ attach=True)  # optional – internet
Traceback (most recent call last):
...
NotImplementedError: you cannot attach a URL
```

```
>>> from sage.all import *
>>> sage.repl.load.load('https://www.sagemath.org/files/loadtest.py', globals(),↪
↪attach=True)  # optional – internet
Traceback (most recent call last):
...
NotImplementedError: you cannot attach a URL
```

The default search path for loading and attaching files is the current working directory, i.e., `'.'`. But you can modify the path with *load_attach_path()*:

```
sage: import tempfile
sage: sage.repl.attach.reset(); reset_load_attach_path()
sage: load_attach_path()
[PosixPath('.')]
sage: with tempfile.TemporaryDirectory() as t_dir:
....:     fname = 'test.py'
....:     fullpath = os.path.join(t_dir, fname)
....:     with open(fullpath, 'w') as f:
....:         _ = f.write("print(37 * 3)")
....:     load_attach_path(t_dir, replace=True)
....:     attach(fname)
111
sage: sage.repl.attach.reset(); reset_load_attach_path() # clean up
```

```
>>> from sage.all import *
>>> import tempfile
>>> sage.repl.attach.reset(); reset_load_attach_path()
>>> load_attach_path()
[PosixPath('.')]
```

3.1. Load Python, Sage, Cython, Fortran and Magma files in Sage

```
>>> with tempfile.TemporaryDirectory() as t_dir:
...     fname = 'test.py'
...     fullpath = os.path.join(t_dir, fname)
...     with open(fullpath, 'w') as f:
...         _ = f.write("print(37 * 3)")
...     load_attach_path(t_dir, replace=True)
...     attach(fname)
111
>>> sage.repl.attach.reset(); reset_load_attach_path() # clean up
```

or by setting the environment variable SAGE_LOAD_ATTACH_PATH to a colon-separated list before starting Sage:

```
$ export SAGE_LOAD_ATTACH_PATH="/path/to/my/library:/path/to/utils"
$ sage
sage: load_attach_path()            # not tested
['.', '/path/to/my/library', '/path/to/utils']
```

sage.repl.load.**load_cython**(*name*)

Helper function to load a Cython file.

INPUT:

- name – filename of the Cython file

OUTPUT:

- A string with Python code to import the names from the compiled module.

sage.repl.load.**load_wrap**(*filename*, *attach=False*)

Encode a load or attach command as valid Python code.

INPUT:

- filename – string or Path object; the argument to the load or attach command

- attach – boolean (default: False); whether to attach filename, instead of loading it

OUTPUT: string

EXAMPLES:

```
sage: sage.repl.load.load_wrap('foo.py', True)
'sage.repl.load.load(sage.repl.load.base64.b64decode("Zm9vLnB5"),globals(),True)'
sage: sage.repl.load.load_wrap('foo.sage')
'sage.repl.load.load(sage.repl.load.base64.b64decode("Zm9vLnNhZ2U="),globals(),
→False)'
sage: m = sage.repl.load.base64.b64decode("Zm9vLnNhZ2U=")
sage: m == b'foo.sage'
True
```

```
>>> from sage.all import *
>>> sage.repl.load.load_wrap('foo.py', True)
'sage.repl.load.load(sage.repl.load.base64.b64decode("Zm9vLnB5"),globals(),True)'
>>> sage.repl.load.load_wrap('foo.sage')
'sage.repl.load.load(sage.repl.load.base64.b64decode("Zm9vLnNhZ2U="),globals(),
→False)'
>>> m = sage.repl.load.base64.b64decode("Zm9vLnNhZ2U=")
```

```
>>> m == b'foo.sage'
True
```

3.2 Keep track of attached files

sage.repl.attach.**add_attached_file**(*filename*)

> Add to the list of attached files.
>
> This is a callback to be used from *load()* after evaluating the attached file the first time.
>
> INPUT:
>
> > • `filename` – string (the fully qualified file name) or `Path` object
>
> EXAMPLES:

```
sage: import sage.repl.attach as af
sage: af.reset()
sage: t = tmp_filename(ext='.py')
sage: af.add_attached_file(t)
sage: af.attached_files()
['/.../tmp_....py']
sage: af.detach(t)
sage: af.attached_files()
[]
```

```
>>> from sage.all import *
>>> import sage.repl.attach as af
>>> af.reset()
>>> t = tmp_filename(ext='.py')
>>> af.add_attached_file(t)
>>> af.attached_files()
['/.../tmp_....py']
>>> af.detach(t)
>>> af.attached_files()
[]
```

sage.repl.attach.**attach**(**files*)

> Attach a file or files to a running instance of Sage and also load that file.

> **ⓘ Note**
>
> Attaching files uses the Python inputhook, which will conflict with other inputhook users. This generally includes GUI main loop integrations, for example tkinter. So you can only use tkinter or attach, but not both at the same time.

> INPUT:
>
> > • `files` – list of filenames (strings) to attach
>
> OUTPUT:
>
> Each file is read in and added to an internal list of watched files. The meaning of reading in a file depends on the file type:

- `.py` files are read in with no preparsing (so, e.g., `2^3` is 2 bit-xor 3);

- `.sage` files are preparsed, then the result is read in;

- `.pyx` files are *not* preparsed, but rather are compiled to a module `m` and then `from m import *` is executed.

The contents of the file are then loaded, which means they are read into the running Sage session. For example, if `foo.sage` contains `x=5`, after attaching `foo.sage` the variable `x` will be set to 5. Moreover, any time you change `foo.sage`, before you execute a command, the attached file will be re-read automatically (with no intervention on your part).

> **↪ See also**
>
> `load()` is the same as `attach()`, but does not automatically reload a file when it changes unless `at-tach=True` is passed.
>
> `%attach` magic can also be used, see `attach()`.

EXAMPLES:

You attach a file, e.g., `foo.sage` or `foo.py` or `foo.pyx`, to a running Sage session by typing:

```
sage: attach('foo.sage')   # not tested
```

```
>>> from sage.all import *
>>> attach('foo.sage')   # not tested
```

Here we test attaching multiple files at once:

```
sage: sage.repl.attach.reset()
sage: t1 = tmp_filename(ext='.py')
sage: with open(t1,'w') as f: _ = f.write("print('hello world')")
sage: t2 = tmp_filename(ext='.py')
sage: with open(t2,'w') as f: _ = f.write("print('hi there xxx')")
sage: attach(t1, t2)
hello world
hi there xxx
sage: af = attached_files(); len(af)
2
sage: t1 in af and t2 in af
True
```

```
>>> from sage.all import *
>>> sage.repl.attach.reset()
>>> t1 = tmp_filename(ext='.py')
>>> with open(t1,'w') as f: _ = f.write("print('hello world')")
>>> t2 = tmp_filename(ext='.py')
>>> with open(t2,'w') as f: _ = f.write("print('hi there xxx')")
>>> attach(t1, t2)
hello world
hi there xxx
>>> af = attached_files(); len(af)
2
>>> t1 in af and t2 in af
True
```

> **See also**
>
> - *attached_files()* returns a list of all currently attached files.
> - *detach()* instructs Sage to remove a file from the internal list of watched files.
> - *load_attach_path()* allows you to get or modify the current search path for loading and attaching files.

sage.repl.attach.**attached_files**()

Return a list of all files attached to the current session with *attach()*.

OUTPUT: the filenames in a sorted list of strings

EXAMPLES:

```
sage: sage.repl.attach.reset()
sage: t = tmp_filename(ext='.py')
sage: with open(t,'w') as f: _ = f.write("print('hello world')")
sage: attach(t)
hello world
sage: af = attached_files(); af
['/....py']
sage: af == [t]
True
```

```
>>> from sage.all import *
>>> sage.repl.attach.reset()
>>> t = tmp_filename(ext='.py')
>>> with open(t,'w') as f: _ = f.write("print('hello world')")
>>> attach(t)
hello world
>>> af = attached_files(); af
['/....py']
>>> af == [t]
True
```

sage.repl.attach.**detach**(*filename*)

Detach a file.

This is the counterpart to *attach()*.

INPUT:

- filename – string, a list of strings or a tuple of strings or a Path, a list of Path or a tuple of Path

EXAMPLES:

```
sage: sage.repl.attach.reset()
sage: t = tmp_filename(ext='.py')
sage: with open(t,'w') as f: _ = f.write("print('hello world')")
sage: attach(t)
hello world
sage: af = attached_files(); len(af)
1
sage: af == [t]
```

```
True
sage: detach(t)
sage: attached_files()
[]

sage: sage.repl.attach.reset(); reset_load_attach_path()
sage: load_attach_path()
[PosixPath('.')]
sage: t_dir = tmp_dir()
sage: fullpath = os.path.join(t_dir, 'test.py')
sage: with open(fullpath, 'w') as f: _ = f.write("print(37 * 3)")
sage: load_attach_path(t_dir, replace=True)
sage: attach('test.py')
111
sage: af = attached_files(); len(af)
1
sage: af == [os.path.normpath(fullpath)]
True
sage: detach('test.py')
sage: attached_files()
[]
sage: attach('test.py')
111
sage: fullpath = os.path.join(t_dir, 'test2.py')
sage: with open(fullpath, 'w') as f: _ = f.write("print(3)")
sage: attach('test2.py')
3
sage: detach(attached_files())
sage: attached_files()
[]
```

```
>>> from sage.all import *
>>> sage.repl.attach.reset()
>>> t = tmp_filename(ext='.py')
>>> with open(t,'w') as f: _ = f.write("print('hello world')")
>>> attach(t)
hello world
>>> af = attached_files(); len(af)
1
>>> af == [t]
True
>>> detach(t)
>>> attached_files()
[]

>>> sage.repl.attach.reset(); reset_load_attach_path()
>>> load_attach_path()
[PosixPath('.')]
>>> t_dir = tmp_dir()
>>> fullpath = os.path.join(t_dir, 'test.py')
>>> with open(fullpath, 'w') as f: _ = f.write("print(37 * 3)")
>>> load_attach_path(t_dir, replace=True)
```

```
>>> attach('test.py')
111
>>> af = attached_files(); len(af)
1
>>> af == [os.path.normpath(fullpath)]
True
>>> detach('test.py')
>>> attached_files()
[]
>>> attach('test.py')
111
>>> fullpath = os.path.join(t_dir, 'test2.py')
>>> with open(fullpath, 'w') as f: _ = f.write("print(3)")
>>> attach('test2.py')
3
>>> detach(attached_files())
>>> attached_files()
[]
```

sage.repl.attach.**load_attach_mode**(*load_debug=None*, *attach_debug=None*)

Get or modify the current debug mode for the behavior of `load()` and *attach()* on `.sage` files.

In debug mode, loaded or attached `.sage` files are preparsed through a file to make their tracebacks more informative. If not in debug mode, then `.sage` files are preparsed in memory only for performance.

At startup, debug mode is `True` for attaching and `False` for loading.

> **ⓘ Note**
>
> This function should really be deprecated and code executed from memory should raise proper tracebacks.

INPUT:

- `load_debug` – boolean or `None` (default); if not `None`, then set a new value for the debug mode for loading files

- `attach_debug` – boolean or `None` (default); same as `load_debug`, but for attaching files

OUTPUT:

If all input values are `None`, returns a tuple giving the current modes for loading and attaching.

EXAMPLES:

```
sage: load_attach_mode()
(False, True)
sage: load_attach_mode(attach_debug=False)
sage: load_attach_mode()
(False, False)
sage: load_attach_mode(load_debug=True)
sage: load_attach_mode()
(True, False)
sage: load_attach_mode(load_debug=False, attach_debug=True)
```

```
>>> from sage.all import *
>>> load_attach_mode()
(False, True)
>>> load_attach_mode(attach_debug=False)
>>> load_attach_mode()
(False, False)
>>> load_attach_mode(load_debug=True)
>>> load_attach_mode()
(True, False)
>>> load_attach_mode(load_debug=False, attach_debug=True)
```

sage.repl.attach.**load_attach_path**(*path=None*, *replace=False*)

> Get or modify the current search path for `load()` and *attach()*.
>
> INPUT:
>
> > • `path` – string or list of strings (default: `None`); path(s) to append to or replace the current path
> >
> > • `replace` – boolean (default: `False`); if `path` is not `None`, whether to *replace* the search path instead of *appending* to it
>
> OUTPUT: none or a *reference* to the current search paths
>
> EXAMPLES:
>
> First, we extend the example given in `load()`'s docstring:

```
sage: sage.repl.attach.reset(); reset_load_attach_path()
sage: load_attach_path()
[PosixPath('.')]
sage: t_dir = tmp_dir()
sage: fullpath = os.path.join(t_dir, 'test.py')
sage: with open(fullpath, 'w') as f:
....:     _ = f.write("print(37 * 3)")
```

```
>>> from sage.all import *
>>> sage.repl.attach.reset(); reset_load_attach_path()
>>> load_attach_path()
[PosixPath('.')]
>>> t_dir = tmp_dir()
>>> fullpath = os.path.join(t_dir, 'test.py')
>>> with open(fullpath, 'w') as f:
...     _ = f.write("print(37 * 3)")
```

> We put a new, empty directory on the attach path for testing (otherwise this will load `test.py` from the current working directory if that happens to exist):

```
sage: import tempfile
sage: with tempfile.TemporaryDirectory() as d:
....:     load_attach_path(d, replace=True)
....:     attach('test.py')
Traceback (most recent call last):
...
OSError: did not find file 'test.py' to load or attach
sage: load_attach_path(t_dir)
```

```
sage: attach('test.py')
111
sage: af = attached_files(); len(af)
1
sage: af == [fullpath]
True
sage: from pathlib import Path
sage: sage.repl.attach.reset(); reset_load_attach_path()
sage: load_attach_path() == [Path('.')]
True
sage: with tempfile.TemporaryDirectory() as d:
....:     load_attach_path(d, replace=True)
....:     load('test.py')
Traceback (most recent call last):
...
OSError: did not find file 'test.py' to load or attach
```

```
>>> from sage.all import *
>>> import tempfile
>>> with tempfile.TemporaryDirectory() as d:
...     load_attach_path(d, replace=True)
...     attach('test.py')
Traceback (most recent call last):
...
OSError: did not find file 'test.py' to load or attach
>>> load_attach_path(t_dir)
>>> attach('test.py')
111
>>> af = attached_files(); len(af)
1
>>> af == [fullpath]
True
>>> from pathlib import Path
>>> sage.repl.attach.reset(); reset_load_attach_path()
>>> load_attach_path() == [Path('.')]
True
>>> with tempfile.TemporaryDirectory() as d:
...     load_attach_path(d, replace=True)
...     load('test.py')
Traceback (most recent call last):
...
OSError: did not find file 'test.py' to load or attach
```

The function returns a reference to the path list:

```
sage: reset_load_attach_path(); load_attach_path()
[PosixPath('.')]
sage: load_attach_path('/path/to/my/sage/scripts'); load_attach_path()
[PosixPath('.'), PosixPath('/path/to/my/sage/scripts')]
sage: load_attach_path(['good', 'bad', 'ugly'], replace=True)
sage: load_attach_path()
[PosixPath('good'), PosixPath('bad'), PosixPath('ugly')]
```

3.2. Keep track of attached files 51

```
sage: p = load_attach_path(); p.pop()
PosixPath('ugly')
sage: p[0] = 'weird'; load_attach_path()
['weird', PosixPath('bad')]
sage: reset_load_attach_path(); load_attach_path()
[PosixPath('.')]
```

```
>>> from sage.all import *
>>> reset_load_attach_path(); load_attach_path()
[PosixPath('.')]
>>> load_attach_path('/path/to/my/sage/scripts'); load_attach_path()
[PosixPath('.'), PosixPath('/path/to/my/sage/scripts')]
>>> load_attach_path(['good', 'bad', 'ugly'], replace=True)
>>> load_attach_path()
[PosixPath('good'), PosixPath('bad'), PosixPath('ugly')]
>>> p = load_attach_path(); p.pop()
PosixPath('ugly')
>>> p[Integer(0)] = 'weird'; load_attach_path()
['weird', PosixPath('bad')]
>>> reset_load_attach_path(); load_attach_path()
[PosixPath('.')]
```

sage.repl.attach.**modified_file_iterator**()

> Iterate over the changed files.
>
> As a side effect the stored time stamps are updated with the actual time stamps. So if you iterate over the attached files in order to reload them and you hit an error then the subsequent files are not marked as read.
>
> Files that are in the process of being saved are excluded.
>
> EXAMPLES:

```
sage: sage.repl.attach.reset()
sage: t = tmp_filename(ext='.py')
sage: attach(t)
sage: from sage.repl.attach import modified_file_iterator
sage: list(modified_file_iterator())
[]
sage: sleep(1)    # filesystem mtime granularity
sage: with open(t, 'w') as f: _ = f.write('1')
sage: list(modified_file_iterator())
[(PosixPath('/.../tmp_....py'), time.struct_time(...))]
```

```
>>> from sage.all import *
>>> sage.repl.attach.reset()
>>> t = tmp_filename(ext='.py')
>>> attach(t)
>>> from sage.repl.attach import modified_file_iterator
>>> list(modified_file_iterator())
[]
>>> sleep(Integer(1))    # filesystem mtime granularity
>>> with open(t, 'w') as f: _ = f.write('1')
>>> list(modified_file_iterator())
```

```
[(PosixPath('/.../tmp_....py'), time.struct_time(...))]
```

sage.repl.attach.**reload_attached_files_if_modified**()

Reload attached files that have been modified.

This is the internal implementation of the attach mechanism.

EXAMPLES:

```
sage: sage.repl.attach.reset()
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: tmp = tmp_filename(ext='.py')
sage: with open(tmp, 'w') as f: _ = f.write('a = 2\n')
sage: shell.run_cell('attach({0})'.format(repr(tmp)))
sage: shell.run_cell('a')
2
sage: sleep(1)    # filesystem mtime granularity
sage: with open(tmp, 'w') as f: _ = f.write('a = 3\n')
```

```
>>> from sage.all import *
>>> sage.repl.attach.reset()
>>> from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell()
>>> tmp = tmp_filename(ext='.py')
>>> with open(tmp, 'w') as f: _ = f.write('a = 2\n')
>>> shell.run_cell('attach({0})'.format(repr(tmp)))
>>> shell.run_cell('a')
2
>>> sleep(Integer(1))    # filesystem mtime granularity
>>> with open(tmp, 'w') as f: _ = f.write('a = 3\n')
```

Note that the doctests are never really at the command prompt where the automatic reload is triggered. So we have to do it manually:

```
sage: shell.run_cell('from sage.repl.attach import reload_attached_files_if_
↪modified')
sage: shell.run_cell('reload_attached_files_if_modified()')
### reloading attached file tmp_....py modified at ... ###

sage: shell.run_cell('a')
3
sage: shell.run_cell('detach({0})'.format(repr(tmp)))
sage: shell.run_cell('attached_files()')
[]
sage: shell.quit()
```

```
>>> from sage.all import *
>>> shell.run_cell('from sage.repl.attach import reload_attached_files_if_modified
↪')
>>> shell.run_cell('reload_attached_files_if_modified()')
### reloading attached file tmp_....py modified at ... ###
```

```
>>> shell.run_cell('a')
3
>>> shell.run_cell('detach({0})'.format(repr(tmp)))
>>> shell.run_cell('attached_files()')
[]
>>> shell.quit()
```

sage.repl.attach.**reset**()

> Remove all the attached files from the list of attached files.

> EXAMPLES:

```
sage: sage.repl.attach.reset()
sage: t = tmp_filename(ext='.py')
sage: with open(t,'w') as f: _ = f.write("print('hello world')")
sage: attach(t)
hello world
sage: af = attached_files(); len(af)
1
sage: af == [t]
True
sage: sage.repl.attach.reset()
sage: attached_files()
[]
```

```
>>> from sage.all import *
>>> sage.repl.attach.reset()
>>> t = tmp_filename(ext='.py')
>>> with open(t,'w') as f: _ = f.write("print('hello world')")
>>> attach(t)
hello world
>>> af = attached_files(); len(af)
1
>>> af == [t]
True
>>> sage.repl.attach.reset()
>>> attached_files()
[]
```

sage.repl.attach.**reset_load_attach_path**()

> Reset the current search path for load() and *attach()*.

> The default path is '.' plus any paths specified in the environment variable SAGE_LOAD_ATTACH_PATH.

> EXAMPLES:

```
sage: load_attach_path()
[PosixPath('.')]
sage: t_dir = tmp_dir()
sage: load_attach_path(t_dir)
sage: from pathlib import Path
sage: Path(t_dir) in load_attach_path()
True
```

```
sage: reset_load_attach_path(); load_attach_path()
[PosixPath('.')]
```

```
>>> from sage.all import *
>>> load_attach_path()
[PosixPath('.')]
>>> t_dir = tmp_dir()
>>> load_attach_path(t_dir)
>>> from pathlib import Path
>>> Path(t_dir) in load_attach_path()
True
>>> reset_load_attach_path(); load_attach_path()
[PosixPath('.')]
```

At startup, Sage adds colon-separated paths in the environment variable SAGE_LOAD_ATTACH_PATH:

```
sage: reset_load_attach_path(); load_attach_path()
[PosixPath('.')]
sage: os.environ['SAGE_LOAD_ATTACH_PATH'] = '/veni/vidi:vici:'
sage: from importlib import reload
sage: reload(sage.repl.attach)     # Simulate startup
<module 'sage.repl.attach' from '...'>
sage: load_attach_path()
[PosixPath('.'), PosixPath('/veni/vidi'), PosixPath('vici')]
sage: del os.environ['SAGE_LOAD_ATTACH_PATH']
sage: reload(sage.repl.preparse)     # Simulate startup
<module 'sage.repl.preparse' from '...'>
sage: reset_load_attach_path(); load_attach_path()
[PosixPath('.')]
```

```
>>> from sage.all import *
>>> reset_load_attach_path(); load_attach_path()
[PosixPath('.')]
>>> os.environ['SAGE_LOAD_ATTACH_PATH'] = '/veni/vidi:vici:'
>>> from importlib import reload
>>> reload(sage.repl.attach)     # Simulate startup
<module 'sage.repl.attach' from '...'>
>>> load_attach_path()
[PosixPath('.'), PosixPath('/veni/vidi'), PosixPath('vici')]
>>> del os.environ['SAGE_LOAD_ATTACH_PATH']
>>> reload(sage.repl.preparse)     # Simulate startup
<module 'sage.repl.preparse' from '...'>
>>> reset_load_attach_path(); load_attach_path()
[PosixPath('.')]
```

PRETTY PRINTING

In addition to making input nicer, we also modify how results are printed. This again builds on how IPython formats output. Technically, this works using a modified displayhook in Python.

4.1 IPython Displayhook Formatters

The classes in this module can be used as IPython displayhook formatters. It has two main features, by default the displayhook contains a new facility for displaying lists of matrices in an easier to read format:

```
sage: [identity_matrix(i) for i in range(2, 5)]                                  #␣
→needs sage.modules
[
                [1 0 0 0]
        [1 0 0]  [0 1 0 0]
[1 0]  [0 1 0]  [0 0 1 0]
[0 1], [0 0 1], [0 0 0 1]
]
```

```
>>> from sage.all import *
>>> [identity_matrix(i) for i in range(Integer(2), Integer(5))]                  ␣
→                # needs sage.modules
[
                [1 0 0 0]
        [1 0 0]  [0 1 0 0]
[1 0]  [0 1 0]  [0 0 1 0]
[0 1], [0 0 1], [0 0 0 1]
]
```

This facility uses `_repr_()` (and a simple string) to try do a nice read format (see `sage.structure.parent.Parent._repr_option()` for details).

With this displayhook there exists an other way for displaying object and more generally, all sage expression as an ASCII art object:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('%display ascii_art')
sage: shell.run_cell('integral(x^2/pi^x, x)')                                    #␣
→needs sage.symbolic
   -x / 2     2                        \
-pi  *\x *log (pi) + 2*x*log(pi) + 2/
```

```
------------------------------------
                3
             log (pi)
sage: shell.run_cell("i = var('i')")                                    #␣
↪needs sage.symbolic
sage: shell.run_cell('sum(i*x^i, i, 0, 10)')                            #␣
↪needs sage.symbolic
    10      9      8      7      6      5      4      3      2
10*x   + 9*x  + 8*x  + 7*x  + 6*x  + 5*x  + 4*x  + 3*x  + 2*x  + x
sage: shell.run_cell('StandardTableaux(4).list()')                      #␣
↪needs sage.combinat
[
[                                                        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 default')
sage: shell.quit()
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell()
>>> shell.run_cell('%display ascii_art')
>>> shell.run_cell('integral(x^2/pi^x, x)')                             #␣
↪needs sage.symbolic
   -x / 2    2                    \
-pi  *\x *log (pi) + 2*x*log(pi) + 2/
------------------------------------
                3
             log (pi)
>>> shell.run_cell("i = var('i')")                                      #␣
↪needs sage.symbolic
>>> shell.run_cell('sum(i*x^i, i, 0, 10)')                              #␣
↪needs sage.symbolic
    10      9      8      7      6      5      4      3      2
10*x   + 9*x  + 8*x  + 7*x  + 6*x  + 5*x  + 4*x  + 3*x  + 2*x  + x
>>> shell.run_cell('StandardTableaux(4).list()')                        #␣
↪needs sage.combinat
[
[                                                        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   ,
<BLANKLINE>
          1 ]
   1  2    2 ]
   3       3 ]
   4   ,   4 ]
>>> shell.run_cell('%display default')
```

```
>>> shell.quit()
```

This other facility uses a simple `AsciiArt` object (see and `sage.structure.sage_object.SageObject._ascii_art_()`).

class sage.repl.display.formatter.**SageDisplayFormatter**(***kwargs: Any*)

> Bases: `DisplayFormatter`
>
> This is where the Sage rich objects are translated to IPython.
>
> INPUT/OUTPUT: see the IPython documentation
>
> EXAMPLES:
>
> This is part of how Sage works with the IPython output system. It cannot be used in doctests:
>
> ```
> sage: from sage.repl.display.formatter import SageDisplayFormatter
> sage: fmt = SageDisplayFormatter()
> Traceback (most recent call last):
> ...
> RuntimeError: check failed: current backend is invalid
> ```
>
> ```
> >>> from sage.all import *
> >>> from sage.repl.display.formatter import SageDisplayFormatter
> >>> fmt = SageDisplayFormatter()
> Traceback (most recent call last):
> ...
> RuntimeError: check failed: current backend is invalid
> ```
>
> **format**(*obj*, *include=None*, *exclude=None*)
>
> > Use the Sage rich output instead of IPython.
> >
> > INPUT/OUTPUT: see the IPython documentation
> >
> > EXAMPLES:
> >
> > ```
> > sage: [identity_matrix(i) for i in range(3,7)]                              #␣
> > ↪needs sage.modules
> > [
> >                                                 [1 0 0 0 0 0]
> >                                  [1 0 0 0 0]    [0 1 0 0 0 0]
> >                     [1 0 0 0]    [0 1 0 0 0]    [0 0 1 0 0 0]
> > [1 0 0]    [0 1 0 0]    [0 0 1 0 0]    [0 0 0 1 0 0]
> > [0 1 0]    [0 0 1 0]    [0 0 0 1 0]    [0 0 0 0 1 0]
> > [0 0 1], [0 0 0 1], [0 0 0 0 1], [0 0 0 0 0 1]
> > ]
> > sage: from sage.repl.interpreter import get_test_shell
> > sage: shell = get_test_shell()
> > sage: shell.run_cell('%display ascii_art')    # indirect doctest
> > sage: shell.run_cell("i = var('i')")                                        #␣
> > ↪needs sage.symbolic
> > sage: shell.run_cell('sum(i*x^i, i, 0, 10)')                                #␣
> > ↪needs sage.symbolic
> >     10       9       8       7       6       5       4       3       2
> > 10*x   + 9*x   + 8*x   + 7*x   + 6*x   + 5*x   + 4*x   + 3*x   + 2*x   + x
> > ```

```
sage: shell.run_cell('%display default')
sage: shell.quit()
```

```
>>> from sage.all import *
>>> [identity_matrix(i) for i in range(Integer(3),Integer(7))]          ␣
↪              # needs sage.modules
[
                                    [1 0 0 0 0 0]
                      [1 0 0 0 0]   [0 1 0 0 0 0]
          [1 0 0 0]   [0 1 0 0 0]   [0 0 1 0 0 0]
[1 0 0]   [0 1 0 0]   [0 0 1 0 0]   [0 0 0 1 0 0]
[0 1 0]   [0 0 1 0]   [0 0 0 1 0]   [0 0 0 0 1 0]
[0 0 1], [0 0 0 1], [0 0 0 0 1], [0 0 0 0 0 1]
]
>>> from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell()
>>> shell.run_cell('%display ascii_art')   # indirect doctest
>>> shell.run_cell("i = var('i')")                                      #␣
↪needs sage.symbolic
>>> shell.run_cell('sum(i*x^i, i, 0, 10)')                              #␣
↪needs sage.symbolic
    10       9       8       7       6       5       4       3       2
10*x    + 9*x   + 8*x   + 7*x   + 6*x   + 5*x   + 4*x   + 3*x   + 2*x   + x
>>> shell.run_cell('%display default')
>>> shell.quit()
```

class sage.repl.display.formatter.**SagePlainTextFormatter**(***kwargs: Any*)

 Bases: `PlainTextFormatter`

 Improved plain text IPython formatter.

 In particular, it correctly print lists of matrices or other objects (see `sage.structure.parent.Parent._repr_option()`).

 > ⚠ **Warning**
 >
 > This IPython formatter is NOT used. You could use it to enable Sage formatting in IPython, but Sage uses its own rich output system that is more flexible and supports different backends.

 INPUT/OUTPUT: see the IPython documentation

 EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.display_formatter.formatters['text/plain']
<IPython.core.formatters.PlainTextFormatter object at 0x...>
sage: shell.quit()
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell()
```

```
>>> shell.display_formatter.formatters['text/plain']
<IPython.core.formatters.PlainTextFormatter object at 0x...>
>>> shell.quit()
```

4.2 The Sage pretty printer

Any transformation to a string goes through here. In other words, the SagePlainTextFormatter is entirely implemented via *SagePrettyPrinter*. Other formatters may or may not use *SagePrettyPrinter* to generate text output.

AUTHORS:

- Bill Cauchois (2009): initial version

- Jean-Baptiste Priez <jbp@kerios.fr> (2013): ASCII art

- Volker Braun (2013): refactored into DisplayHookBase

class sage.repl.display.pretty_print.**SagePrettyPrinter**(*output*, *max_width*, *newline*, *max_seq_length=None*)

Bases: PrettyPrinter

Pretty print Sage objects for the commandline.

INPUT:

See IPython documentation.

EXAMPLES:

```
sage: 123
123
```

```
>>> from sage.all import *
>>> Integer(123)
123
```

IPython pretty printers:

```
sage: set({1, 2, 3})
{1, 2, 3}
sage: dict(zzz=123, aaa=99, xab=10)     # sorted by keys
{'aaa': 99, 'xab': 10, 'zzz': 123}
```

```
>>> from sage.all import *
>>> set({Integer(1), Integer(2), Integer(3)})
{1, 2, 3}
>>> dict(zzz=Integer(123), aaa=Integer(99), xab=Integer(10))     # sorted by keys
{'aaa': 99, 'xab': 10, 'zzz': 123}
```

These are overridden in IPython in a way that we feel is somewhat confusing, and we prefer to print them like plain Python which is more informative. See Issue #14466

```
sage: 'this is a string'
'this is a string'
sage: type(123)
```

```
<class 'sage.rings.integer.Integer'>
sage: type
<... 'type'>
sage: import types
sage: type('name', (), {})
<class '__main__.name'>
sage: types.BuiltinFunctionType
<class 'builtin_function_or_method'>

sage: def foo(): pass
sage: foo
<function foo at 0x...>
```

```
>>> from sage.all import *
>>> 'this is a string'
'this is a string'
>>> type(Integer(123))
<class 'sage.rings.integer.Integer'>
>>> type
<... 'type'>
>>> import types
>>> type('name', (), {})
<class '__main__.name'>
>>> types.BuiltinFunctionType
<class 'builtin_function_or_method'>

>>> def foo(): pass
>>> foo
<function foo at 0x...>
```

DEBUG = False

pretty(*obj*)

> Pretty print `obj`.
>
> This is the only method that outside code should invoke.
>
> INPUT:
>
> - `obj` – anything
>
> OUTPUT: string representation for object
>
> EXAMPLES:

```
sage: from sage.repl.display.pretty_print import SagePrettyPrinter
sage: from io import StringIO
sage: stream = StringIO()
sage: SagePrettyPrinter(stream, 78, '\n').pretty([type, 123, 'foo'])
sage: stream.getvalue()
"[<... 'type'>,"
```

```
>>> from sage.all import *
>>> from sage.repl.display.pretty_print import SagePrettyPrinter
```

```
>>> from io import StringIO
>>> stream = StringIO()
>>> SagePrettyPrinter(stream, Integer(78), '\n').pretty([type, Integer(123),
→'foo'])
>>> stream.getvalue()
"[<... 'type'>,"
```

pretty_repr = (TallListRepr pretty printer, LargeMatrixHelpRepr pretty printer, SomeIPythonRepr pretty printer, PlainPythonRepr pretty printer)

toplevel()

> Return whether we are currently at the top level.
>
> OUTPUT:
>
> boolean; whether we are currently pretty-printing an object at the outermost level (`True`), or whether the object is inside a container (`False`).
>
> EXAMPLES:

```
sage: from sage.repl.display.pretty_print import SagePrettyPrinter
sage: from io import StringIO
sage: stream = StringIO()
sage: spp = SagePrettyPrinter(stream, 78, '\n')
sage: spp.toplevel()
True
```

```
>>> from sage.all import *
>>> from sage.repl.display.pretty_print import SagePrettyPrinter
>>> from io import StringIO
>>> stream = StringIO()
>>> spp = SagePrettyPrinter(stream, Integer(78), '\n')
>>> spp.toplevel()
True
```

4.3 Representations of objects

class sage.repl.display.fancy_repr.**LargeMatrixHelpRepr**

> Bases: *ObjectReprABC*
>
> Representation including help for large Sage matrices.
>
> **__call__**(*obj*, *p*, *cycle*)
>
> > Format matrix.
> >
> > INPUT:
> >
> > - `obj` – anything; object to format
> > - `p` – PrettyPrinter instance
> > - `cycle` – boolean; whether there is a cycle
> >
> > OUTPUT:
> >
> > boolean; whether the representer is applicable to `obj`. If `True`, the string representation is appended to `p`.

EXAMPLES:

```
sage: # needs sage.modules
sage: from sage.repl.display.fancy_repr import LargeMatrixHelpRepr
sage: M = identity_matrix(40)
sage: pp = LargeMatrixHelpRepr()
sage: pp.format_string(M)
"40 x 40 dense matrix over Integer Ring (use the '.str()' method...)"
sage: pp.format_string([M, M])
'--- object not handled by representer ---'
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> from sage.repl.display.fancy_repr import LargeMatrixHelpRepr
>>> M = identity_matrix(Integer(40))
>>> pp = LargeMatrixHelpRepr()
>>> pp.format_string(M)
"40 x 40 dense matrix over Integer Ring (use the '.str()' method...)"
>>> pp.format_string([M, M])
'--- object not handled by representer ---'
```

Leads to:

```
sage: M                                                                    #␣
↪needs sage.modules
40 x 40 dense matrix over Integer Ring (use the '.str()' method...)
sage: [M, M]                                                               #␣
↪needs sage.modules
[40 x 40 dense matrix over Integer Ring,
 40 x 40 dense matrix over Integer Ring]
```

```
>>> from sage.all import *
>>> M                                                                      #␣
↪needs sage.modules
40 x 40 dense matrix over Integer Ring (use the '.str()' method...)
>>> [M, M]                                                                 #␣
↪needs sage.modules
[40 x 40 dense matrix over Integer Ring,
 40 x 40 dense matrix over Integer Ring]
```

class sage.repl.display.fancy_repr.**ObjectReprABC**

Bases: `object`

The abstract base class of an object representer.

__call__(*obj*, *p*, *cycle*)

Format object.

INPUT:

- `obj` – anything; object to format

- `p` – PrettyPrinter instance

- `cycle` – boolean; whether there is a cycle

OUTPUT:

boolean; whether the representer is applicable to `obj`. If `True`, the string representation is appended to `p`.

EXAMPLES:

```
sage: from sage.repl.display.fancy_repr import ObjectReprABC
sage: ObjectReprABC().format_string(123)   # indirect doctest
'Error: ObjectReprABC.__call__ is abstract'
```

```
>>> from sage.all import *
>>> from sage.repl.display.fancy_repr import ObjectReprABC
>>> ObjectReprABC().format_string(Integer(123))   # indirect doctest
'Error: ObjectReprABC.__call__ is abstract'
```

format_string(*obj*)

For doctesting only: Directly return string.

INPUT:

- `obj` – anything; object to format

OUTPUT: string

EXAMPLES:

```
sage: from sage.repl.display.fancy_repr import ObjectReprABC
sage: ObjectReprABC().format_string(123)
'Error: ObjectReprABC.__call__ is abstract'
```

```
>>> from sage.all import *
>>> from sage.repl.display.fancy_repr import ObjectReprABC
>>> ObjectReprABC().format_string(Integer(123))
'Error: ObjectReprABC.__call__ is abstract'
```

class sage.repl.display.fancy_repr.**PlainPythonRepr**

Bases: *ObjectReprABC*

The ordinary Python representation.

__call__(*obj*, *p*, *cycle*)

Format object.

INPUT:

- `obj` – anything; object to format
- `p` – PrettyPrinter instance
- `cycle` – boolean; whether there is a cycle

OUTPUT:

boolean; whether the representer is applicable to `obj`. If `True`, the string representation is appended to `p`.

EXAMPLES:

```
sage: from sage.repl.display.fancy_repr import PlainPythonRepr
sage: pp = PlainPythonRepr()
sage: pp.format_string(type(1))
"<class 'sage.rings.integer.Integer'>"
```

```
>>> from sage.all import *
>>> from sage.repl.display.fancy_repr import PlainPythonRepr
>>> pp = PlainPythonRepr()
>>> pp.format_string(type(Integer(1)))
"<class 'sage.rings.integer.Integer'>"
```

Do not swallow a trailing newline at the end of the output of a custom representer. Note that it is undesirable to have a trailing newline, and if we don't display it you can't fix it:

```
sage: class Newline():
....:     def __repr__(self):
....:         return 'newline\n'
sage: n = Newline()
sage: pp.format_string(n)
'newline\n'
sage: pp.format_string([n, n, n])
'[newline\n, newline\n, newline\n]'
sage: [n, n, n]
[newline
, newline
, newline
]
```

```
>>> from sage.all import *
>>> class Newline():
...     def __repr__(self):
...         return 'newline\n'
>>> n = Newline()
>>> pp.format_string(n)
'newline\n'
>>> pp.format_string([n, n, n])
'[newline\n, newline\n, newline\n]'
>>> [n, n, n]
[newline
, newline
, newline
]
```

class sage.repl.display.fancy_repr.**SomeIPythonRepr**

Bases: *ObjectReprABC*

Some selected representers from IPython.

EXAMPLES:

```
sage: from sage.repl.display.fancy_repr import SomeIPythonRepr
sage: SomeIPythonRepr()
SomeIPythonRepr pretty printer
```

```
>>> from sage.all import *
>>> from sage.repl.display.fancy_repr import SomeIPythonRepr
>>> SomeIPythonRepr()
SomeIPythonRepr pretty printer
```

__call__(*obj*, *p*, *cycle*)

 Format object.

 INPUT:

- `obj` – anything; object to format

- `p` – PrettyPrinter instance

- `cycle` – boolean; whether there is a cycle

 OUTPUT:

 boolean; whether the representer is applicable to `obj`. If `True`, the string representation is appended to `p`.

 EXAMPLES:

```
sage: from sage.repl.display.fancy_repr import SomeIPythonRepr
sage: pp = SomeIPythonRepr()
sage: pp.format_string(set([1, 2, 3]))
'{1, 2, 3}'
```

```
>>> from sage.all import *
>>> from sage.repl.display.fancy_repr import SomeIPythonRepr
>>> pp = SomeIPythonRepr()
>>> pp.format_string(set([Integer(1), Integer(2), Integer(3)]))
'{1, 2, 3}'
```

class sage.repl.display.fancy_repr.**TallListRepr**

 Bases: *ObjectReprABC*

 Special representation for lists with tall entries (e.g. matrices).

 __call__(*obj*, *p*, *cycle*)

  Format list/tuple.

  INPUT:

- `obj` – anything; object to format

- `p` – PrettyPrinter instance

- `cycle` – boolean; whether there is a cycle

  OUTPUT:

  boolean; whether the representer is applicable to `obj`. If `True`, the string representation is appended to `p`.

  EXAMPLES:

```
sage: from sage.repl.display.fancy_repr import TallListRepr
sage: format_list = TallListRepr().format_string
sage: format_list([1, 2, identity_matrix(2)])                      #␣
↪needs sage.modules
'[\n      [1 0]\n1, 2, [0 1]\n]'
```

```
>>> from sage.all import *
>>> from sage.repl.display.fancy_repr import TallListRepr
>>> format_list = TallListRepr().format_string
>>> format_list([Integer(1), Integer(2), identity_matrix(Integer(2))])     ␣
```

```
↪                               # needs sage.modules
'[\n      [1 0]\n1, 2, [0 1]\n]'
```

Check that Issue #18743 is fixed:

```
sage: class Foo():
....:     def __repr__(self):
....:         return '''BBB     AA    RRR
....: B  B  A  A  R  R
....: BBB   AAAA  RRR
....: B  B  A  A  R  R
....: BBB   A  A  R   R'''
....:     def _repr_option(self, key):
....:         return key == 'ascii_art'
sage: F = Foo()
sage: [F, F]
[
BBB     AA    RRR     BBB     AA    RRR
B  B  A  A  R  R   B  B  A  A  R  R
BBB   AAAA  RRR     BBB   AAAA  RRR
B  B  A  A  R  R   B  B  A  A  R  R
BBB   A  A  R   R, BBB   A  A  R   R
]
```

```
>>> from sage.all import *
>>> class Foo():
...     def __repr__(self):
...         return '''BBB     AA    RRR
... B  B  A  A  R  R
... BBB   AAAA  RRR
... B  B  A  A  R  R
... BBB   A  A  R   R'''
...     def _repr_option(self, key):
...         return key == 'ascii_art'
>>> F = Foo()
>>> [F, F]
[
BBB     AA    RRR     BBB     AA    RRR
B  B  A  A  R  R   B  B  A  A  R  R
BBB   AAAA  RRR     BBB   AAAA  RRR
B  B  A  A  R  R   B  B  A  A  R  R
BBB   A  A  R   R, BBB   A  A  R   R
]
```

4.4 Utility functions for pretty-printing

These utility functions are used in the implementations of _repr_ methods elsewhere.

class sage.repl.display.util.**TallListFormatter**

Bases: object

Special representation for lists with tall entries (e.g. matrices).

__call__(*the_list*)

>   Return "tall list" string representation.
>
>   See also *try_format()*.
>
>   INPUT:
>
>   - `the_list` – list or tuple
>
>   OUTPUT: string
>
>   EXAMPLES:

```
sage: from sage.repl.display.util import format_list
sage: format_list(['not', 'tall'])
"['not', 'tall']"
```

```
>>> from sage.all import *
>>> from sage.repl.display.util import format_list
>>> format_list(['not', 'tall'])
"['not', 'tall']"
```

MAX_COLUMN = 70

try_format(*the_list*)

>   First check whether a list is "tall" – whether the reprs of the elements of the list will span multiple lines and
>   cause the list to be printed awkwardly. If not, this function returns `None` and does nothing; you should revert
>   back to the normal method for printing an object (its repr). If so, return the string in the special format. Note
>   that the special format isn't just for matrices. Any object with a multiline repr will be formatted.
>
>   INPUT:
>
>   - `the_list` – the list (or a tuple)
>
>   OUTPUT:
>
>   String or `None`. The latter is returned if the list is not deemed to be tall enough and another formatter should
>   be used.

DISPLAY BACKEND INFRASTRUCTURE

5.1 Display Manager

This is the heart of the rich output system, the display manager arbitrates between

- Backend capabilities: what can be displayed

- Backend preferences: what gives good quality on the backend

- Sage capabilities: every Sage object can only generate certain representations, and

- User preferences: typeset vs. plain text vs. ascii art, etc.

The display manager is a singleton class, Sage always has exactly one instance of it. Use `get_display_manager()` to obtain it.

EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager();  dm
The Sage display manager using the doctest backend
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output import get_display_manager
>>> dm = get_display_manager();  dm
The Sage display manager using the doctest backend
```

exception sage.repl.rich_output.display_manager.**DisplayException**

Bases: `Exception`

Base exception for all rich output-related exceptions.

EXAMPLES:

```
sage: from sage.repl.rich_output.display_manager import DisplayException
sage: raise DisplayException('foo')
Traceback (most recent call last):
...
DisplayException: foo
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.display_manager import DisplayException
>>> raise DisplayException('foo')
Traceback (most recent call last):
```

```
...
DisplayException: foo
```

class sage.repl.rich_output.display_manager.**DisplayManager**

Bases: `SageObject`

The Display Manager.

Used to decide what kind of rich output is best.

EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: get_display_manager()
The Sage display manager using the doctest backend
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output import get_display_manager
>>> get_display_manager()
The Sage display manager using the doctest backend
```

check_backend_class(*backend_class*)

Check that the current backend is an instance of `backend_class`.

This is, for example, used by the Sage IPython display formatter to ensure that the IPython backend is in use.

INPUT:

- `backend_class` – type of a backend class

OUTPUT:

This method returns nothing. A `RuntimeError` is raised if `backend_class` is not the type of the current backend.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendSimple
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: dm.check_backend_class(BackendSimple)
Traceback (most recent call last):
...
RuntimeError: check failed: current backend is invalid
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_base import BackendSimple
>>> from sage.repl.rich_output import get_display_manager
>>> dm = get_display_manager()
>>> dm.check_backend_class(BackendSimple)
Traceback (most recent call last):
...
RuntimeError: check failed: current backend is invalid
```

display_immediately(*obj*, ***rich_repr_kwds*)

Show output without going back to the command line prompt.

This method must be called to create rich output from an object when we are not returning to the command line prompt, for example during program execution. Typically, it is being called by `sage.plot.graphics.Graphics.show()`.

INPUT:

- `obj` – anything; the object to be shown

- `rich_repr_kwds` – optional keyword arguments that are passed through to `obj._rich_repr_`

EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: dm.display_immediately(1/2)
1/2
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output import get_display_manager
>>> dm = get_display_manager()
>>> dm.display_immediately(Integer(1)/Integer(2))
1/2
```

displayhook(*obj*)

Implementation of the displayhook.

Every backend must pass the value of the last statement of a line / cell to this method. See also *display_immediately()* if you want do display rich output while a program is running.

INPUT:

- `obj` – anything; the object to be shown

OUTPUT: whatever the backend's displayhook method returned

EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: dm.displayhook(1/2)
1/2
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output import get_display_manager
>>> dm = get_display_manager()
>>> dm.displayhook(Integer(1)/Integer(2))
1/2
```

classmethod get_instance()

Get the singleton instance.

This class method is equivalent to `get_display_manager()`.

OUTPUT: the display manager singleton

EXAMPLES:

```
sage: from sage.repl.rich_output.display_manager import DisplayManager
sage: DisplayManager.get_instance()
The Sage display manager using the doctest backend
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.display_manager import DisplayManager
>>> DisplayManager.get_instance()
The Sage display manager using the doctest backend
```

graphics_from_save(*save_function*, *save_kwds*, *file_extension*, *output_container*, *figsize=None*, *dpi=None*)

Helper to construct graphics.

This method can be used to simplify the implementation of a _rich_repr_ method of a graphics object if there is already a function to save graphics to a file.

INPUT:

- save_function – callable that can save graphics to a file and accepts options like `sage.plot.graphics.Graphics.save()`

- save_kwds – dictionary; keyword arguments that are passed to the save function

- file_extension – string starting with '.'; the file extension of the graphics file

- output_container – subclass of `sage.repl.rich_output.output_basic.OutputBase`. The output container to use. Must be one of the types in `supported_output()`.

- figsize – pair of integers (optional). The desired graphics size in pixels. Suggested, but need not be respected by the output.

- dpi – integer (optional). The desired resolution in dots per inch. Suggested, but need not be respected by the output.

OUTPUT: an instance of output_container

EXAMPLES:

```
sage: # needs sage.plot sage.symbolic
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: plt = plot(sin)
sage: out = dm.graphics_from_save(plt.save, dict(), '.png',
....:                             dm.types.OutputImagePng)
sage: out
OutputImagePng container
sage: out.png.get().startswith(b'\x89PNG')
True
sage: out.png.filename()    # random
'/home/user/.sage/temp/localhost.localdomain/23903/tmp_pu5woK.png'
```

```
>>> from sage.all import *
>>> # needs sage.plot sage.symbolic
>>> from sage.repl.rich_output import get_display_manager
>>> dm = get_display_manager()
>>> plt = plot(sin)
>>> out = dm.graphics_from_save(plt.save, dict(), '.png',
...                             dm.types.OutputImagePng)
```

```
>>> out
OutputImagePng container
>>> out.png.get().startswith(b'\x89PNG')
True
>>> out.png.filename()    # random
'/home/user/.sage/temp/localhost.localdomain/23903/tmp_pu5woK.png'
```

is_in_terminal()

    Test whether the UI is meant to run in a terminal.

    When this method returns `True`, you can assume that it is possible to use `raw_input` or launch external programs that take over the input.

    Otherwise, you should assume that the backend runs remotely or in a pty controlled by another program. Then you should not launch external programs with a (text or graphical) UI.

    This is used to enable/disable interpreter consoles.

    OUTPUT:

    Boolean.

property preferences

    Return the preferences.

    OUTPUT:

    The display preferences as instance of *DisplayPreferences*.

    EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: dm.preferences
Display preferences:
* align_latex is not specified
* graphics is not specified
* supplemental_plot = never
* text is not specified
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output import get_display_manager
>>> dm = get_display_manager()
>>> dm.preferences
Display preferences:
* align_latex is not specified
* graphics is not specified
* supplemental_plot = never
* text is not specified
```

supported_output()

    Return the output container classes that can be used.

    OUTPUT:

    Frozen set of subclasses of *OutputBase*. If the backend defines derived container classes, this method will always return their base classes.

EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: dm.types.OutputPlainText in dm.supported_output()
True
sage: type(dm.supported_output())
<... 'frozenset'>
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output import get_display_manager
>>> dm = get_display_manager()
>>> dm.types.OutputPlainText in dm.supported_output()
True
>>> type(dm.supported_output())
<... 'frozenset'>
```

switch_backend(*backend*, ***kwds*)

Switch to a new backend.

INPUT:

- backend – instance of *BackendBase*

- kwds – optional keyword arguments that are passed on to the *install()* method

OUTPUT: the previous backend

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendSimple
sage: simple = BackendSimple()
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager();  dm
The Sage display manager using the doctest backend

sage: previous = dm.switch_backend(simple)
sage: dm
The Sage display manager using the simple backend
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_base import BackendSimple
>>> simple = BackendSimple()
>>> from sage.repl.rich_output import get_display_manager
>>> dm = get_display_manager();  dm
The Sage display manager using the doctest backend

>>> previous = dm.switch_backend(simple)
>>> dm
The Sage display manager using the simple backend
```

Restore the doctest backend:

```
sage: dm.switch_backend(previous) is simple
True
```

```
>>> from sage.all import *
>>> dm.switch_backend(previous) is simple
True
```

threejs_scripts(*online*)

Return Three.js script tag for the current backend.

INPUT:

- `online` – boolean determining script usage context

OUTPUT: string containing script tag

> **ⓘ Note**
>
> This base method handles `online=True` case only, serving CDN script tag. Location of script for offline usage is backend-specific.

EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: get_display_manager().threejs_scripts(online=True)                    #␣
↪needs sage.plot
'...<script src="https://cdn.jsdelivr.net/gh/sagemath/threejs-sage@...'
sage: get_display_manager().threejs_scripts(online=False)                   #␣
↪needs sage.plot
Traceback (most recent call last):
...
ValueError: current backend does not support
offline threejs graphics
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output import get_display_manager
>>> get_display_manager().threejs_scripts(online=True)                      #␣
↪needs sage.plot
'...<script src="https://cdn.jsdelivr.net/gh/sagemath/threejs-sage@...'
>>> get_display_manager().threejs_scripts(online=False)                     #␣
↪needs sage.plot
Traceback (most recent call last):
...
ValueError: current backend does not support
offline threejs graphics
```

property types

Catalog of all output container types.

Note that every output type must be registered in *sage.repl.rich_output.output_catalog*.

OUTPUT:

Returns the *sage.repl.rich_output.output_catalog* module.

EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: dm.types.OutputPlainText
<class 'sage.repl.rich_output.output_basic.OutputPlainText'>
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output import get_display_manager
>>> dm = get_display_manager()
>>> dm.types.OutputPlainText
<class 'sage.repl.rich_output.output_basic.OutputPlainText'>
```

exception sage.repl.rich_output.display_manager.**OutputTypeException**

> Bases: *DisplayException*
>
> Wrong Output container.
>
> The output containers are the subclasses of *OutputBase* that contain the entire output. The display backends must create output containers of a suitable type depending on the displayed Python object. This exception indicates that there is a mistake in the backend and it returned the wrong type of output container.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.display_manager import OutputTypeException
sage: raise OutputTypeException('foo')
Traceback (most recent call last):
...
OutputTypeException: foo
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.display_manager import OutputTypeException
>>> raise OutputTypeException('foo')
Traceback (most recent call last):
...
OutputTypeException: foo
```

exception sage.repl.rich_output.display_manager.**RichReprWarning**

> Bases: UserWarning
>
> Warning that is throws if a call to _rich_repr_ fails.
>
> If an object implements _rich_repr_ then it must return a value, possibly None to indicate that no rich output can be generated. But it may not raise an exception as it is very confusing for the user if the displayhook fails.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.display_manager import RichReprWarning
sage: raise RichReprWarning('foo')
Traceback (most recent call last):
...
RichReprWarning: foo
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.display_manager import RichReprWarning
>>> raise RichReprWarning('foo')
Traceback (most recent call last):
```

```
...
RichReprWarning: foo
```

class sage.repl.rich_output.display_manager.**restricted_output**(*display_manager*, *output_classes*)

    Bases: object

    Context manager to temporarily restrict the accepted output types.

    In the context, the output is restricted to the output container types listed in output_classes. Additionally, display preferences are changed not to show graphics.

    INPUT:

        • display_manager – the display manager

        • output_classes – iterable of output container types

    EXAMPLES:

```
sage: from sage.repl.rich_output.display_manager import (
....:     get_display_manager, restricted_output)
sage: dm = get_display_manager()
sage: restricted_output(dm, [dm.types.OutputPlainText])
<sage.repl.rich_output.display_manager.restricted_output object at 0x...>
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.display_manager import (
...     get_display_manager, restricted_output)
>>> dm = get_display_manager()
>>> restricted_output(dm, [dm.types.OutputPlainText])
<sage.repl.rich_output.display_manager.restricted_output object at 0x...>
```

5.2 Display Preferences

This class is used to express display preferences that are not simply a choice of a particular output format. For example, whether to prefer vector over raster graphics. By convention, the value None is always a valid value for a preference and means no particular preference.

EXAMPLES:

```
sage: from sage.repl.rich_output.preferences import DisplayPreferences
sage: prefs = DisplayPreferences()
sage: prefs.available_options()
(align_latex, graphics, supplemental_plot, text)
sage: prefs.text is None
True
sage: prefs.text = 'ascii_art'
sage: prefs.text
'ascii_art'
sage: prefs
Display preferences:
* align_latex is not specified
* graphics is not specified
* supplemental_plot is not specified
* text = ascii_art
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.preferences import DisplayPreferences
>>> prefs = DisplayPreferences()
>>> prefs.available_options()
(align_latex, graphics, supplemental_plot, text)
>>> prefs.text is None
True
>>> prefs.text = 'ascii_art'
>>> prefs.text
'ascii_art'
>>> prefs
Display preferences:
* align_latex is not specified
* graphics is not specified
* supplemental_plot is not specified
* text = ascii_art
```

Properties can be unset by deleting them or by assigning `None`:

```
sage: prefs.text = 'ascii_art'
sage: del prefs.text
sage: prefs.text is None
True

sage: prefs.text = 'ascii_art'
sage: prefs.text = None
sage: prefs.text is None
True
```

```
>>> from sage.all import *
>>> prefs.text = 'ascii_art'
>>> del prefs.text
>>> prefs.text is None
True

>>> prefs.text = 'ascii_art'
>>> prefs.text = None
>>> prefs.text is None
True
```

Properties have documentation attached:

```
sage: import pydoc
sage: doc = pydoc.render_doc(prefs)
sage: assert ' graphics' in doc
sage: assert '    Preferred graphics format' in doc
sage: assert ' text' in doc
sage: assert '    Which textual representation is preferred' in doc
```

```
>>> from sage.all import *
>>> import pydoc
>>> doc = pydoc.render_doc(prefs)
>>> assert ' graphics' in doc
```

```
>>> assert '      Preferred graphics format' in doc
>>> assert ' text' in doc
>>> assert '      Which textual representation is preferred' in doc
```

Values can also be specified as keyword arguments to the constructor:

```
sage: DisplayPreferences(text='latex')
Display preferences:
* align_latex is not specified
* graphics is not specified
* supplemental_plot is not specified
* text = latex
```

```
>>> from sage.all import *
>>> DisplayPreferences(text='latex')
Display preferences:
* align_latex is not specified
* graphics is not specified
* supplemental_plot is not specified
* text = latex
```

> ✏️ **Todo**
>
> A value-checking preference system should be used elsewhere in Sage, too. The class here is just a simple implementation, a proper implementation would use a metaclass to construct the preference items.

class sage.repl.rich_output.preferences.**DisplayPreferences**(*args*, ***kwds*)

  Bases: *PreferencesABC*

  property align_latex

  Preferred mode of latex displays

  Allowed values:

  - `None` (default): no preference
  - 'center'
  - 'left'

  property graphics

  Preferred graphics format

  Allowed values:

  - `None` (default): no preference
  - 'disable'
  - 'vector'
  - 'raster'

  property supplemental_plot

  Whether to graphically display graphs and other graph-like objects that implement rich output. When not specified small objects are show graphically and large objects as textual overview.

Allowed values:

- `None` (default): no preference

- 'always'

- 'never'

property text

Which textual representation is preferred

Allowed values:

- `None` (default): no preference

- 'plain'

- 'ascii_art'

- 'unicode_art'

- 'latex'

class sage.repl.rich_output.preferences.**PreferencesABC**(**args*, ***kwds*)

Bases: `SageObject`

Preferences for displaying graphics.

These can be preferences expressed by the user or by the display backend. They are specified as keyword arguments.

INPUT:

- `*args*` – positional arguments are preferences instances. The property values will be inherited from left to right, that is, later parents override values from earlier parents.

- `**kwds` – keyword arguments; will be used to initialize properties, and override inherited values if necessary

EXAMPLES:

```
sage: from sage.repl.rich_output.preferences import DisplayPreferences
sage: p1 = DisplayPreferences(graphics='vector')
sage: p2 = DisplayPreferences(graphics='raster')
sage: DisplayPreferences(p1, p2)
Display preferences:
* align_latex is not specified
* graphics = raster
* supplemental_plot is not specified
* text is not specified
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.preferences import DisplayPreferences
>>> p1 = DisplayPreferences(graphics='vector')
>>> p2 = DisplayPreferences(graphics='raster')
>>> DisplayPreferences(p1, p2)
Display preferences:
* align_latex is not specified
* graphics = raster
* supplemental_plot is not specified
* text is not specified
```

If specified in the opposite order, the setting from `p1` is inherited:

```
sage: DisplayPreferences(p2, p1)
Display preferences:
* align_latex is not specified
* graphics = vector
* supplemental_plot is not specified
* text is not specified
```

```
>>> from sage.all import *
>>> DisplayPreferences(p2, p1)
Display preferences:
* align_latex is not specified
* graphics = vector
* supplemental_plot is not specified
* text is not specified
```

Further keywords override:

```
sage: DisplayPreferences(p2, p1, graphics='disable')
Display preferences:
* align_latex is not specified
* graphics = disable
* supplemental_plot is not specified
* text is not specified
```

```
>>> from sage.all import *
>>> DisplayPreferences(p2, p1, graphics='disable')
Display preferences:
* align_latex is not specified
* graphics = disable
* supplemental_plot is not specified
* text is not specified
```

available_options()

> Return the available options.
>
> OUTPUT:
>
> tuple of the preference items as instances of *Property*
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.rich_output.preferences import DisplayPreferences
> sage: DisplayPreferences().available_options()
> (align_latex, graphics, supplemental_plot, text)
> ```
>
> ```
> >>> from sage.all import *
> >>> from sage.repl.rich_output.preferences import DisplayPreferences
> >>> DisplayPreferences().available_options()
> (align_latex, graphics, supplemental_plot, text)
> ```

class sage.repl.rich_output.preferences.**Property**(*name*, *allowed_values*, *doc=None*)

> Bases: *property*
>
> Preference item.

INPUT:

- `name` – string; the name of the property

- `allowed_values` – list/tuple/iterable of allowed values

- `doc` – string (optional); the docstring of the property

EXAMPLES:

```
sage: from sage.repl.rich_output.preferences import Property
sage: prop = Property('foo', [0, 1, 2], 'The Foo Property')
sage: prop.__doc__
'The Foo Property\n\nAllowed values:\n\n* ``None`` (default): no preference\n\n*␣
→0\n\n* 1\n\n* 2'
sage: prop.allowed_values
(0, 1, 2)
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.preferences import Property
>>> prop = Property('foo', [Integer(0), Integer(1), Integer(2)], 'The Foo Property
→')
>>> prop.__doc__
'The Foo Property\n\nAllowed values:\n\n* ``None`` (default): no preference\n\n*␣
→0\n\n* 1\n\n* 2'
>>> prop.allowed_values
(0, 1, 2)
```

deleter(*prefs*)

Delete the current value of the property.

INPUT:

- `prefs` – the *PreferencesABC* instance that the property is bound to

EXAMPLES:

```
sage: from sage.repl.rich_output.preferences import Property, PreferencesABC
sage: prop = Property('foo', [0, 1, 2], 'The Foo Property')
sage: prefs = PreferencesABC()
sage: prop.getter(prefs) is None
True
sage: prop.setter(prefs, 1)
sage: prop.deleter(prefs)
sage: prop.getter(prefs) is None
True
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.preferences import Property, PreferencesABC
>>> prop = Property('foo', [Integer(0), Integer(1), Integer(2)], 'The Foo␣
→Property')
>>> prefs = PreferencesABC()
>>> prop.getter(prefs) is None
True
>>> prop.setter(prefs, Integer(1))
>>> prop.deleter(prefs)
```

```
>>> prop.getter(prefs) is None
True
```

getter(*prefs*)

> Get the current value of the property.
>
> INPUT:
>
> • prefs – the *PreferencesABC* instance that the property is bound to
>
> OUTPUT: one of the allowed values or None if not set
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.preferences import Property, PreferencesABC
sage: prop = Property('foo', [0, 1, 2], 'The Foo Property')
sage: prefs = PreferencesABC()
sage: prop.getter(prefs) is None
True
sage: prop.setter(prefs, 1)
sage: prop.getter(prefs)
1
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.preferences import Property, PreferencesABC
>>> prop = Property('foo', [Integer(0), Integer(1), Integer(2)], 'The Foo␣
↪Property')
>>> prefs = PreferencesABC()
>>> prop.getter(prefs) is None
True
>>> prop.setter(prefs, Integer(1))
>>> prop.getter(prefs)
1
```

setter(*prefs*, *value*)

> Get the current value of the property.
>
> INPUT:
>
> • prefs – the *PreferencesABC* instance that the property is bound to
>
> • value – anything. The new value of the property. Setting a property to None is equivalent to deleting the value.
>
> OUTPUT:
>
> This method does not return anything. A ValueError is raised if the given value is not one of the allowed values.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.preferences import Property, PreferencesABC
sage: prop = Property('foo', [0, 1, 2], 'The Foo Property')
sage: prefs = PreferencesABC()
sage: prop.getter(prefs) is None
True
sage: prop.setter(prefs, 1)
```

```
sage: prop.getter(prefs)
1

sage: prop.setter(prefs, None)
sage: prop.getter(prefs) is None
True
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.preferences import Property, PreferencesABC
>>> prop = Property('foo', [Integer(0), Integer(1), Integer(2)], 'The Foo␣
↪Property')
>>> prefs = PreferencesABC()
>>> prop.getter(prefs) is None
True
>>> prop.setter(prefs, Integer(1))
>>> prop.getter(prefs)
1

>>> prop.setter(prefs, None)
>>> prop.getter(prefs) is None
True
```

5.3 The `pretty_print` command

Works similar to the `print` function, except that it always tries to use a rich output for an object, as specified via the text display preference. If such a rich output is not available, it falls back on the plain text.

EXAMPLES:

```
sage: pretty_print(1, 2, 3)
1 2 3

sage: pretty_print(x^2 / (x + 1))                                    #␣
↪needs sage.symbolic
x^2/(x + 1)
```

```
>>> from sage.all import *
>>> pretty_print(Integer(1), Integer(2), Integer(3))
1 2 3

>>> pretty_print(x**Integer(2) / (x + Integer(1)))                     ␣
↪                # needs sage.symbolic
x^2/(x + 1)
```

EXAMPLES:

```
sage: %display ascii_art  # not tested
sage: pretty_print(x^2 / (x + 1))                                    #␣
↪needs sage.symbolic
  2
 x
```

```
-----
x + 1
```

```
>>> from sage.all import *
>>> %display ascii_art  # not tested
>>> pretty_print(x**Integer(2) / (x + Integer(1)))                              ␣
↪                # needs sage.symbolic
  2
  x
-----
x + 1
```

EXAMPLES:

Printing a graphics object just prints a string, whereas *pretty_print()* does not print anything and just shows the graphics instead:

```
sage: print(plot(sin))                                                          #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
sage: pretty_print(plot(sin))                                                   #␣
↪needs sage.plot sage.symbolic
```

```
>>> from sage.all import *
>>> print(plot(sin))                                                            #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
>>> pretty_print(plot(sin))                                                     #␣
↪needs sage.plot sage.symbolic
```

class sage.repl.rich_output.pretty_print.**SequencePrettyPrinter**(*args*, ***kwds*)

Bases: `SageObject`

Pretty Printer for Multiple Arguments.

INPUT/OUTPUT:

Same as *pretty_print()*, except that the number of arguments must be >= 2. Otherwise its not a sequence of things to print.

EXAMPLES:

```
sage: pretty_print(1, 2, 3)    # indirect doctest
1 2 3
sage: from sage.repl.rich_output.pretty_print import SequencePrettyPrinter
sage: SequencePrettyPrinter(1, 2, 3).pretty_print()
1 2 3
```

```
>>> from sage.all import *
>>> pretty_print(Integer(1), Integer(2), Integer(3))    # indirect doctest
1 2 3
>>> from sage.repl.rich_output.pretty_print import SequencePrettyPrinter
>>> SequencePrettyPrinter(Integer(1), Integer(2), Integer(3)).pretty_print()
1 2 3
```

is_homogeneous(*common_type*)

> Return whether the pretty print items are homogeneous.
>
> INPUT:
>
> - `common_type` – a type
>
> OUTPUT:
>
> boolean; whether all items to be pretty printed are of said type.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.rich_output.pretty_print import SequencePrettyPrinter
> sage: seq = SequencePrettyPrinter(1, 2, 3)
> sage: seq.is_homogeneous(Integer)
> True
> sage: seq.is_homogeneous(str)
> False
> ```
>
> ```
> >>> from sage.all import *
> >>> from sage.repl.rich_output.pretty_print import SequencePrettyPrinter
> >>> seq = SequencePrettyPrinter(Integer(1), Integer(2), Integer(3))
> >>> seq.is_homogeneous(Integer)
> True
> >>> seq.is_homogeneous(str)
> False
> ```

pretty_print()

> Actually do the pretty print.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.rich_output.pretty_print import SequencePrettyPrinter
> sage: SequencePrettyPrinter(1, 2, 3).pretty_print()
> 1 2 3
> ```
>
> ```
> >>> from sage.all import *
> >>> from sage.repl.rich_output.pretty_print import SequencePrettyPrinter
> >>> SequencePrettyPrinter(Integer(1), Integer(2), Integer(3)).pretty_print()
> 1 2 3
> ```
>
> The keyword arguments are only used the first time graphics output is generated:
>
> ```
> sage: seq = SequencePrettyPrinter(Graph(), Graph(), edge_labels=True)        #␣
> ↪needs sage.graphs sage.plot
> sage: seq.pretty_print()   # does not pass edge_labels to graphics object    #␣
> ↪needs sage.graphs sage.plot
> sage: seq._concatenate_graphs().show(edge_labels=True)                       #␣
> ↪needs sage.graphs sage.plot
> Traceback (most recent call last):
> ...
> TypeError: ...matplotlib() got an unexpected keyword argument 'edge_labels'...
> ```

```
>>> from sage.all import *
>>> seq = SequencePrettyPrinter(Graph(), Graph(), edge_labels=True)          #␣
↪needs sage.graphs sage.plot
>>> seq.pretty_print()   # does not pass edge_labels to graphics object    #␣
↪needs sage.graphs sage.plot
>>> seq._concatenate_graphs().show(edge_labels=True)                        #␣
↪needs sage.graphs sage.plot
Traceback (most recent call last):
...
TypeError: ...matplotlib() got an unexpected keyword argument 'edge_labels'...
```

sage.repl.rich_output.pretty_print.**pretty_print**(**args*, ***kwds*)

Pretty print the arguments using rich output if available.

This function is similar to `print()`, except that a rich output representation such as `ascii_art` or Latex is printed instead of the string representation.

Note that the output depends on the global display preferences specified via *preferences()*. If the display preference for `text` is not specified, Latex output is preferred.

For graphical objects, a graphical output is used.

For certain homogeneous multiple positional arguments, a suitable combined graphical output is generated. In particular, graphs and plots are treated special. Otherwise this function concatenates the textual representations.

INPUT:

- `*args` – any number of positional arguments; the objects to pretty print

- `**kwds` – optional keyword arguments that are passed to the rich representation. Examples include:

  - `dpi` – dots per inch

  - `figsize`- [width, height] (same for square aspect)

  - `axes` – (default: `True`)

  - `fontsize` – positive integer

  - `frame` – boolean (default: `False`); draw a MATLAB-like frame around the image

EXAMPLES:

```
sage: pretty_print(ZZ)
Integer Ring

sage: pretty_print("Integers = ", ZZ) # trac 11775
'Integers = ' Integer Ring
```

```
>>> from sage.all import *
>>> pretty_print(ZZ)
Integer Ring

>>> pretty_print("Integers = ", ZZ) # trac 11775
'Integers = ' Integer Ring
```

To typeset LaTeX code as-is, use `LatexExpr`:

```
sage: pretty_print(LatexExpr(r"\frac{x^2 + 1}{x - 2}"))
\frac{x^2 + 1}{x - 2}
```

```
>>> from sage.all import *
>>> pretty_print(LatexExpr(r"\frac{x^2 + 1}{x - 2}"))
\frac{x^2 + 1}{x - 2}
```

For text-based backends, the default text display preference is to output plain text which is usually the same as using `print()`:

```
sage: pretty_print(x^2 / (x + 1))                                              #␣
↪needs sage.symbolic
x^2/(x + 1)

sage: t = BinaryTrees(3).first()                                               #␣
↪needs sage.graphs
sage: pretty_print(t)                                                          #␣
↪needs sage.graphs
[., [., [., .]]]
sage: print(t)                                                                 #␣
↪needs sage.graphs
[., [., [., .]]]
```

```
>>> from sage.all import *
>>> pretty_print(x**Integer(2) / (x + Integer(1)))                             ␣
↪                 # needs sage.symbolic
x^2/(x + 1)

>>> t = BinaryTrees(Integer(3)).first()                                        ␣
↪        # needs sage.graphs
>>> pretty_print(t)                                                            #␣
↪needs sage.graphs
[., [., [., .]]]
>>> print(t)                                                                   #␣
↪needs sage.graphs
[., [., [., .]]]
```

EXAMPLES:

Changing the text display preference affects the output of this function. The following illustrates a possible use-case:

```
sage: %display ascii_art  # not tested
sage: for t in BinaryTrees(3)[:3]:                                             #␣
↪needs sage.graphs
....:     pretty_print(t)
o
 \
  o
   \
    o
o
 \
  o
```

(continues on next page)

```
 /
o
  o
 / \
o   o

sage: pretty_print(x^2 / (x + 1))                                    #␣
↪needs sage.symbolic
   2
  x
-----
x + 1
```

```
>>> from sage.all import *
>>> %display ascii_art  # not tested
>>> for t in BinaryTrees(Integer(3))[:Integer(3)]:                   ␣
↪                # needs sage.graphs
...     pretty_print(t)
o
 \
   o
    \
     o
o
 \
   o
 /
o
   o
 / \
o   o

>>> pretty_print(x**Integer(2) / (x + Integer(1)))                  ␣
↪                # needs sage.symbolic
   2
  x
-----
x + 1
```

sage.repl.rich_output.pretty_print.**show**(*args*, ***kwds*)

> Alias for pretty_print.
>
> This function is an alias for *pretty_print()*.
>
> INPUT/OUTPUT:
>
> See *pretty_print()*. Except if the argument is a graph, in which case it is plotted instead.
>
> EXAMPLES:

```
sage: show(1)
1
```

```
>>> from sage.all import *
>>> show(Integer(1))
1
```

5.4 Output Buffer

This is the fundamental unit of rich output, a single immutable buffer (either in-memory or as a file). Rich output always consists of one or more buffers. Ideally, the Sage library always uses the buffer object as an in-memory buffer. But you can also ask it for a filename, and it will save the data to a file if necessary. Either way, the buffer object presents the same interface for getting the content of an in-memory buffer or a temporary file. So any rich output backends do not need to know where the buffer content is actually stored.

EXAMPLES:

```
sage: from sage.repl.rich_output.buffer import OutputBuffer
sage: buf = OutputBuffer('this is the buffer content');  buf
buffer containing 26 bytes
sage: buf.get().decode('ascii')
'this is the buffer content'
sage: type(buf.get()) is bytes
True
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.buffer import OutputBuffer
>>> buf = OutputBuffer('this is the buffer content');  buf
buffer containing 26 bytes
>>> buf.get().decode('ascii')
'this is the buffer content'
>>> type(buf.get()) is bytes
True
```

class sage.repl.rich_output.buffer.**OutputBuffer**(*data*)

    Bases: SageObject

    Data stored either in memory or as a file.

    This class is an abstraction for "files", in that they can either be defined by a bytes array (Python 3) or string (Python 2) or by a file (see *from_file()*).

    INPUT:

        • data – bytes; the data that is stored in the buffer

    EXAMPLES:

```
sage: from sage.repl.rich_output.buffer import OutputBuffer
sage: buf = OutputBuffer('this is the buffer content');  buf
buffer containing 26 bytes

sage: buf2 = OutputBuffer(buf);  buf2
buffer containing 26 bytes

sage: buf.get_str()
'this is the buffer content'
```

```
sage: buf.filename(ext='.txt')
'/....txt'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.buffer import OutputBuffer
>>> buf = OutputBuffer('this is the buffer content');  buf
buffer containing 26 bytes

>>> buf2 = OutputBuffer(buf);  buf2
buffer containing 26 bytes

>>> buf.get_str()
'this is the buffer content'
>>> buf.filename(ext='.txt')
'/....txt'
```

filename(*ext=None*)

> Return the filename.
>
> INPUT:
>
> > • `ext` – string; the file extension
>
> OUTPUT:
>
> Name of a file, most likely a temporary file. If `ext` is specified, the filename will have that extension.
>
> You must not modify the returned file. Its permissions are set to readonly to help with that.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.buffer import OutputBuffer
sage: buf = OutputBuffer('test')
sage: buf.filename()                 # random output
'/home/user/.sage/temp/hostname/26085/tmp_RNSfAc'

sage: os.path.isfile(buf.filename())
True
sage: buf.filename(ext='txt')  # random output
'/home/user/.sage/temp/hostname/26085/tmp_Rjjp4V.txt'
sage: buf.filename(ext='txt').endswith('.txt')
True
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.buffer import OutputBuffer
>>> buf = OutputBuffer('test')
>>> buf.filename()               # random output
'/home/user/.sage/temp/hostname/26085/tmp_RNSfAc'

>>> os.path.isfile(buf.filename())
True
>>> buf.filename(ext='txt')  # random output
'/home/user/.sage/temp/hostname/26085/tmp_Rjjp4V.txt'
>>> buf.filename(ext='txt').endswith('.txt')
True
```

classmethod from_file(*filename*)

Construct buffer from data in file.

> ⚠️ **Warning**
>
> The buffer assumes that the file content remains the same during the lifetime of the Sage session. To communicate this to the user, the file permissions will be changed to read only.

INPUT:

- `filename` – string; the filename under which the data is stored

OUTPUT: string containing the buffer data

EXAMPLES:

```
sage: from sage.repl.rich_output.buffer import OutputBuffer
sage: name = sage.misc.temporary_file.tmp_filename()
sage: with open(name, 'wb') as f:
....:     _ = f.write(b'file content')
sage: buf = OutputBuffer.from_file(name);  buf
buffer containing 12 bytes

sage: buf.filename() == name
True
sage: buf.get_str()
'file content'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.buffer import OutputBuffer
>>> name = sage.misc.temporary_file.tmp_filename()
>>> with open(name, 'wb') as f:
...     _ = f.write(b'file content')
>>> buf = OutputBuffer.from_file(name);  buf
buffer containing 12 bytes

>>> buf.filename() == name
True
>>> buf.get_str()
'file content'
```

get()

Return the buffer content.

OUTPUT: bytes; string in Python 2.x

EXAMPLES:

```
sage: from sage.repl.rich_output.buffer import OutputBuffer
sage: c = OutputBuffer('test1234').get(); c.decode('ascii')
'test1234'
sage: type(c) is bytes
True
sage: c = OutputBuffer('été').get()
```

```
sage: type(c) is bytes
True
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.buffer import OutputBuffer
>>> c = OutputBuffer('test1234').get(); c.decode('ascii')
'test1234'
>>> type(c) is bytes
True
>>> c = OutputBuffer('été').get()
>>> type(c) is bytes
True
```

get_str()

Return the buffer content as a `str` object for the current Python version.

That is, returns a Python 2-style encoding-agnostic `str` on Python 2, and returns a unicode `str` on Python 3 with the buffer content decoded from UTF-8. In other words, this is equivalent to `OutputBuffer.get` on Python 2 and `OutputBuffer.get_unicode` on Python 3. This is useful in some cases for cross-compatible code.

OUTPUT: a `str` object

EXAMPLES:

```
sage: from sage.repl.rich_output.buffer import OutputBuffer
sage: c = OutputBuffer('test1234').get_str(); c
'test1234'
sage: type(c) is str
True
sage: c = OutputBuffer('été').get_str()
sage: type(c) is str
True
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.buffer import OutputBuffer
>>> c = OutputBuffer('test1234').get_str(); c
'test1234'
>>> type(c) is str
True
>>> c = OutputBuffer('été').get_str()
>>> type(c) is str
True
```

get_unicode()

Return the buffer content as string.

OUTPUT:

String. Unicode in Python 2.x. Raises a `UnicodeEncodeError` if the data is not valid utf-8.

EXAMPLES:

```
sage: from sage.repl.rich_output.buffer import OutputBuffer
sage: OutputBuffer('test1234').get().decode('ascii')
'test1234'
sage: OutputBuffer('test1234').get_unicode()
'test1234'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.buffer import OutputBuffer
>>> OutputBuffer('test1234').get().decode('ascii')
'test1234'
>>> OutputBuffer('test1234').get_unicode()
'test1234'
```

save_as(*filename*)

Save a copy of the buffer content.

You may edit the returned file, unlike the file returned by *filename()*.

INPUT:

- `filename` – string; the file name to save under

EXAMPLES:

```
sage: from sage.repl.rich_output.buffer import OutputBuffer
sage: buf = OutputBuffer('test')
sage: buf.filename(ext='txt')   # random output
sage: tmp = tmp_dir()
sage: filename = os.path.join(tmp, 'foo.txt')
sage: buf.save_as(filename)
sage: with open(filename, 'r') as f:
....:     f.read()
'test'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.buffer import OutputBuffer
>>> buf = OutputBuffer('test')
>>> buf.filename(ext='txt')   # random output
>>> tmp = tmp_dir()
>>> filename = os.path.join(tmp, 'foo.txt')
>>> buf.save_as(filename)
>>> with open(filename, 'r') as f:
...     f.read()
'test'
```

5.5 Basic Output Types

The Sage rich representation system requires a special container class to hold the data for each type of rich output. They all inherit from *OutputBase*, though a more typical example is *OutputPlainText*. Some output classes consist of more than one data buffer, for example jmol or certain animation formats. The output class is independent of user preferences and of the display backend.

The display backends can define derived classes to attach backend-specific display functionality to, for example how to launch a viewer. But they must not change how the output container is created. To enforce this, the Sage _rich_repr_

magic method will only ever see the output class defined here. The display manager will promote it to a backend-specific subclass if necessary prior to displaying it.

To create new types of output, you must create your own subclass of *OutputBase* and register it in *sage.repl.rich_output.output_catalog*.

> ⚠️ **Warning**
>
> All rich output data in subclasses of *OutputBase* must be contained in *OutputBuffer* instances. You must never reference any files on the local file system, as there is no guarantee that the notebook server and the worker process are on the same computer. Or even share a common file system.

class sage.repl.rich_output.output_basic.**OutputAsciiArt**(*ascii_art*)

> Bases: *OutputBase*
>
> ASCII Art Output.
>
> INPUT:
>
> - ascii_art – *OutputBuffer*. Alternatively, a string (bytes) can be passed directly which will then be converted into an *OutputBuffer*. Ascii art rendered into a string.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.rich_output.output_catalog import OutputAsciiArt
> sage: OutputAsciiArt(':-}')
> OutputAsciiArt container
> ```
>
> ```
> >>> from sage.all import *
> >>> from sage.repl.rich_output.output_catalog import OutputAsciiArt
> >>> OutputAsciiArt(':-}')
> OutputAsciiArt container
> ```
>
> **classmethod example**()
>
> > Construct a sample ascii art output container.
> >
> > This static method is meant for doctests, so they can easily construct an example.
> >
> > OUTPUT: an instance of *OutputAsciiArt*
> >
> > EXAMPLES:
> >
> > ```
> > sage: from sage.repl.rich_output.output_catalog import OutputAsciiArt
> > sage: OutputAsciiArt.example()
> > OutputAsciiArt container
> > sage: OutputAsciiArt.example().ascii_art.get_str()
> > '[                        *    *    *    * ]\n[      **    **    *    *    *    *    *␣
> > ↪     *  ]\n[ ***, * , *  , **, ** , *, * , * , *    ]'
> > ```
> >
> > ```
> > >>> from sage.all import *
> > >>> from sage.repl.rich_output.output_catalog import OutputAsciiArt
> > >>> OutputAsciiArt.example()
> > OutputAsciiArt container
> > >>> OutputAsciiArt.example().ascii_art.get_str()
> > '[                        *    *    *    * ]\n[      **    **    *    *    *    *    *␣
> > ↪     *  ]\n[ ***, * , *  , **, ** , *, * , * , *    ]'
> > ```

print_to_stdout()

> Write the data to stdout.
>
> This is just a convenience method to help with debugging.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputAsciiArt
sage: ascii_art = OutputAsciiArt.example()
sage: ascii_art.print_to_stdout()
[                        *   *   *     * ]
[       **   **   *     *  *   *  *     * ]
[ ***, * , *  , **, ** , *, * , * , *    ]
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputAsciiArt
>>> ascii_art = OutputAsciiArt.example()
>>> ascii_art.print_to_stdout()
[                        *   *   *     * ]
[       **   **   *     *  *   *  *     * ]
[ ***, * , *  , **, ** , *, * , * , *    ]
```

class sage.repl.rich_output.output_basic.**OutputBase**

> Bases: `SageObject`
>
> Base class for all rich output containers.
>
> **classmethod example**()
>
> > Construct a sample instance.
> >
> > This static method is meant for doctests, so they can easily construct an example.
> >
> > OUTPUT: an instance of the *OutputBase* subclass
> >
> > EXAMPLES:

```
sage: from sage.repl.rich_output.output_basic import OutputBase
sage: OutputBase.example()
Traceback (most recent call last):
...
NotImplementedError: derived classes must implement this class method
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_basic import OutputBase
>>> OutputBase.example()
Traceback (most recent call last):
...
NotImplementedError: derived classes must implement this class method
```

class sage.repl.rich_output.output_basic.**OutputLatex**(*latex*)

> Bases: *OutputBase*
>
> LaTeX Output.

> **ⓘ Note**
>
> The LaTeX commands will only use a subset of LaTeX that can be displayed by MathJax.

INPUT:

- latex – *OutputBuffer*. Alternatively, a string (bytes) can be passed directly which will then be converted into an *OutputBuffer*. String containing the latex equation code. Excludes the surrounding dollar signs / LaTeX equation environment.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputLatex
sage: OutputLatex(latex(sqrt(x)))                                            #␣
↪needs sage.symbolic
OutputLatex container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputLatex
>>> OutputLatex(latex(sqrt(x)))                                            # needs␣
↪sage.symbolic
OutputLatex container
```

display_equation()

> Return the LaTeX code for a display equation.
>
> OUTPUT: string
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.rich_output.output_catalog import OutputLatex
> sage: rich_output = OutputLatex('1')
> sage: rich_output.latex
> buffer containing 1 bytes
> sage: rich_output.latex.get_str()
> '1'
> sage: rich_output.display_equation()
> '\\begin{equation}\n1\n\\end{equation}'
> ```
>
> ```
> >>> from sage.all import *
> >>> from sage.repl.rich_output.output_catalog import OutputLatex
> >>> rich_output = OutputLatex('1')
> >>> rich_output.latex
> buffer containing 1 bytes
> >>> rich_output.latex.get_str()
> '1'
> >>> rich_output.display_equation()
> '\\begin{equation}\n1\n\\end{equation}'
> ```

classmethod example()

> Construct a sample LaTeX output container.
>
> This static method is meant for doctests, so they can easily construct an example.
>
> OUTPUT: an instance of *OutputLatex*

5.5. Basic Output Types   **99**

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputLatex
sage: OutputLatex.example()
OutputLatex container
sage: OutputLatex.example().latex.get_str()
'\\newcommand{\\Bold}[1]{\\mathbf{#1}}\\int \\sin\\left(x\\right)\\,{d x}'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputLatex
>>> OutputLatex.example()
OutputLatex container
>>> OutputLatex.example().latex.get_str()
'\\newcommand{\\Bold}[1]{\\mathbf{#1}}\\int \\sin\\left(x\\right)\\,{d x}'
```

inline_equation()

Return the LaTeX code for an inline equation.

OUTPUT: string

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputLatex
sage: rich_output = OutputLatex('1')
sage: rich_output.latex
buffer containing 1 bytes
sage: rich_output.latex.get_str()
'1'
sage: rich_output.inline_equation()
'\\begin{math}\n1\n\\end{math}'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputLatex
>>> rich_output = OutputLatex('1')
>>> rich_output.latex
buffer containing 1 bytes
>>> rich_output.latex.get_str()
'1'
>>> rich_output.inline_equation()
'\\begin{math}\n1\n\\end{math}'
```

print_to_stdout()

Write the data to stdout.

This is just a convenience method to help with debugging.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputLatex
sage: rich_output = OutputLatex.example()
sage: rich_output.print_to_stdout()
\newcommand{\Bold}[1]{\mathbf{#1}}\int \sin\left(x\right)\,{d x}
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputLatex
```

```
>>> rich_output = OutputLatex.example()
>>> rich_output.print_to_stdout()
\newcommand{\Bold}[1]{\mathbf{#1}}\int \sin\left(x\right)\,{d x}
```

class sage.repl.rich_output.output_basic.**OutputPlainText**(*plain_text*)

> Bases: *OutputBase*
>
> Plain Text Output.
>
> INPUT:
>
> - plain_text – *OutputBuffer*. Alternatively, a bytes (string in Python 2.x) or string (unicode in Python 2.x) can be passed directly which will then be converted into an *OutputBuffer*. The plain text output.
>
> This should always be exactly the same as the (non-rich) output from the _repr_ method. Every backend object must support plain text output as fallback.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.rich_output.output_catalog import OutputPlainText
> sage: OutputPlainText('foo')
> OutputPlainText container
> ```
>
> ```
> >>> from sage.all import *
> >>> from sage.repl.rich_output.output_catalog import OutputPlainText
> >>> OutputPlainText('foo')
> OutputPlainText container
> ```
>
> **classmethod example**()
>
> > Construct a sample plain text output container.
> >
> > This static method is meant for doctests, so they can easily construct an example.
> >
> > OUTPUT: an instance of *OutputPlainText*
> >
> > EXAMPLES:
> >
> > ```
> > sage: from sage.repl.rich_output.output_catalog import OutputPlainText
> > sage: OutputPlainText.example()
> > OutputPlainText container
> > sage: OutputPlainText.example().text.get_str()
> > 'Example plain text output'
> > ```
> >
> > ```
> > >>> from sage.all import *
> > >>> from sage.repl.rich_output.output_catalog import OutputPlainText
> > >>> OutputPlainText.example()
> > OutputPlainText container
> > >>> OutputPlainText.example().text.get_str()
> > 'Example plain text output'
> > ```
>
> **print_to_stdout**()
>
> > Write the data to stdout.
> >
> > This is just a convenience method to help with debugging.
> >
> > EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputPlainText
sage: plain_text = OutputPlainText.example()
sage: plain_text.print_to_stdout()
Example plain text output
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputPlainText
>>> plain_text = OutputPlainText.example()
>>> plain_text.print_to_stdout()
Example plain text output
```

class sage.repl.rich_output.output_basic.**OutputUnicodeArt**(*unicode_art*)

> Bases: *OutputBase*
>
> Unicode Art Output.
>
> Similar to *OutputAsciiArt* but using the entire unicode range.
>
> INPUT:
>
> - unicode_art – *OutputBuffer*. Alternatively, a string (unicode in Python 2.x) can be passed directly which will then be converted into an *OutputBuffer*. Unicode art rendered into a string.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputUnicodeArt
sage: OutputUnicodeArt(u':-}')
OutputUnicodeArt container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputUnicodeArt
>>> OutputUnicodeArt(u':-}')
OutputUnicodeArt container
```

> **classmethod example**()
>
> > Construct a sample unicode art output container.
> >
> > This static method is meant for doctests, so they can easily construct an example.
> >
> > OUTPUT: an instance of *OutputUnicodeArt*
> >
> > EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputUnicodeArt
sage: OutputUnicodeArt.example()
OutputUnicodeArt container
sage: print(OutputUnicodeArt.example().unicode_art.get_unicode())
⎛-11   0   1⎞
⎜  3  -1   0⎟
⎝ -1  -1   0⎠
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputUnicodeArt
>>> OutputUnicodeArt.example()
OutputUnicodeArt container
>>> print(OutputUnicodeArt.example().unicode_art.get_unicode())
```

```
⎛-11   0   1⎞
|  3  -1   0|
⎝ -1  -1   0⎠
```

print_to_stdout()

>   Write the data to stdout.

>   This is just a convenience method to help with debugging.

>   EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputUnicodeArt
sage: unicode_art = OutputUnicodeArt.example()
sage: unicode_art.print_to_stdout()
⎛-11   0   1⎞
|  3  -1   0|
⎝ -1  -1   0⎠
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputUnicodeArt
>>> unicode_art = OutputUnicodeArt.example()
>>> unicode_art.print_to_stdout()
⎛-11   0   1⎞
|  3  -1   0|
⎝ -1  -1   0⎠
```

5.6 Graphics Output Types

This module defines the rich output types for 2-d images, both vector and raster graphics.

class sage.repl.rich_output.output_graphics.**OutputImageDvi**(*dvi*)

>   Bases: *OutputBase*

>   DVI Image.

>   INPUT:

>   - dvi – *OutputBuffer*. Alternatively, a string (bytes) can be passed directly which will then be converted
>   into an *OutputBuffer*. The DVI data.

>   EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImageDvi
sage: OutputImageDvi.example()        # indirect doctest
OutputImageDvi container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputImageDvi
>>> OutputImageDvi.example()        # indirect doctest
OutputImageDvi container
```

>   **classmethod example**()

>   >   Construct a sample DVI output container.

>   >   This static method is meant for doctests, so they can easily construct an example.

OUTPUT: an instance of *OutputImageDvi*

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImageDvi
sage: OutputImageDvi.example()
OutputImageDvi container
sage: OutputImageDvi.example().dvi
buffer containing 212 bytes
sage: b'TeX output' in OutputImageDvi.example().dvi.get()
True
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputImageDvi
>>> OutputImageDvi.example()
OutputImageDvi container
>>> OutputImageDvi.example().dvi
buffer containing 212 bytes
>>> b'TeX output' in OutputImageDvi.example().dvi.get()
True
```

class sage.repl.rich_output.output_graphics.**OutputImageGif**(*gif*)

Bases: *OutputBase*

GIF Image (possibly animated).

INPUT:

- `gif` – *OutputBuffer*. Alternatively, a string (bytes) can be passed directly which will then be converted into an *OutputBuffer*. The GIF image data.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImageGif
sage: OutputImageGif.example()     # indirect doctest
OutputImageGif container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputImageGif
>>> OutputImageGif.example()     # indirect doctest
OutputImageGif container
```

classmethod example()

Construct a sample GIF output container.

This static method is meant for doctests, so they can easily construct an example.

OUTPUT: an instance of *OutputImageGif*

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImageGif
sage: OutputImageGif.example()
OutputImageGif container
sage: OutputImageGif.example().gif
buffer containing 408 bytes
```

(continues on next page)

```
sage: OutputImageGif.example().gif.get().startswith(b'GIF89a')
True
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputImageGif
>>> OutputImageGif.example()
OutputImageGif container
>>> OutputImageGif.example().gif
buffer containing 408 bytes
>>> OutputImageGif.example().gif.get().startswith(b'GIF89a')
True
```

html_fragment()

> Return a self-contained HTML fragment displaying the image.
>
> This is a workaround for the Jupyter notebook which doesn't support GIF directly.
>
> OUTPUT: string. HTML fragment for displaying the GIF image
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImageGif
sage: OutputImageGif.example().html_fragment()
'<img src="data:image/gif;base64,R0lGOD1...zd3t/g4eLj5OVDQQA7"/>'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputImageGif
>>> OutputImageGif.example().html_fragment()
'<img src="data:image/gif;base64,R0lGOD1...zd3t/g4eLj5OVDQQA7"/>'
```

class sage.repl.rich_output.output_graphics.**OutputImageJpg**(*jpg*)

> Bases: *OutputBase*
>
> JPEG Image.
>
> INPUT:
>
> - jpg – *OutputBuffer*. Alternatively, a string (bytes) can be passed directly which will then be converted into an *OutputBuffer*. The JPEG image data.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImageJpg
sage: OutputImageJpg.example()    # indirect doctest
OutputImageJpg container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputImageJpg
>>> OutputImageJpg.example()    # indirect doctest
OutputImageJpg container
```

classmethod example()

> Construct a sample JPEG output container.
>
> This static method is meant for doctests, so they can easily construct an example.

OUTPUT: an instance of *OutputImageJpg*

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImageJpg
sage: OutputImageJpg.example()
OutputImageJpg container
sage: OutputImageJpg.example().jpg
buffer containing 978 bytes
sage: OutputImageJpg.example().jpg.get().startswith(b'\xff\xd8\xff\xe0\x00\
↪x10JFIF')
True
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputImageJpg
>>> OutputImageJpg.example()
OutputImageJpg container
>>> OutputImageJpg.example().jpg
buffer containing 978 bytes
>>> OutputImageJpg.example().jpg.get().startswith(b'\xff\xd8\xff\xe0\x00\
↪x10JFIF')
True
```

class sage.repl.rich_output.output_graphics.**OutputImagePdf**(*pdf*)

Bases: *OutputBase*

PDF Image.

INPUT:

- pdf – *OutputBuffer*. Alternatively, a string (bytes) can be passed directly which will then be converted into an *OutputBuffer*. The PDF data.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImagePdf
sage: OutputImagePdf.example()    # indirect doctest
OutputImagePdf container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputImagePdf
>>> OutputImagePdf.example()    # indirect doctest
OutputImagePdf container
```

classmethod example()

Construct a sample PDF output container.

This static method is meant for doctests, so they can easily construct an example.

OUTPUT: an instance of *OutputImagePdf*

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImagePdf
sage: OutputImagePdf.example()
OutputImagePdf container
sage: OutputImagePdf.example().pdf
```

```
buffer containing 4285 bytes
sage: OutputImagePdf.example().pdf.get().startswith(b'%PDF-1.4')
True
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputImagePdf
>>> OutputImagePdf.example()
OutputImagePdf container
>>> OutputImagePdf.example().pdf
buffer containing 4285 bytes
>>> OutputImagePdf.example().pdf.get().startswith(b'%PDF-1.4')
True
```

class sage.repl.rich_output.output_graphics.**OutputImagePng**(*png*)

Bases: *OutputBase*

PNG Image.

> **ⓘ Note**
>
> Every backend that is capable of displaying any kind of graphics is supposed to support the PNG format at least.

INPUT:

- png – *OutputBuffer*. Alternatively, a string (bytes) can be passed directly which will then be converted into an *OutputBuffer*. The PNG image data.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImagePng
sage: OutputImagePng.example()    # indirect doctest
OutputImagePng container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputImagePng
>>> OutputImagePng.example()    # indirect doctest
OutputImagePng container
```

classmethod example()

Construct a sample PNG output container.

This static method is meant for doctests, so they can easily construct an example.

OUTPUT: an instance of *OutputImagePng*

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImagePng
sage: OutputImagePng.example()
OutputImagePng container
sage: OutputImagePng.example().png
buffer containing 608 bytes
```

```
sage: OutputImagePng.example().png.get().startswith(b'\x89PNG')
True
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputImagePng
>>> OutputImagePng.example()
OutputImagePng container
>>> OutputImagePng.example().png
buffer containing 608 bytes
>>> OutputImagePng.example().png.get().startswith(b'\x89PNG')
True
```

class sage.repl.rich_output.output_graphics.**OutputImageSvg**(*svg*)

> Bases: *OutputBase*
>
> SVG Image.
>
> INPUT:
>
> > • svg – *OutputBuffer*. Alternatively, a string (bytes) can be passed directly which will then be converted
> > into an *OutputBuffer*. The SVG image data.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImageSvg
sage: OutputImageSvg.example()    # indirect doctest
OutputImageSvg container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputImageSvg
>>> OutputImageSvg.example()    # indirect doctest
OutputImageSvg container
```

> **classmethod example**()
>
> > Construct a sample SVG output container.
> >
> > This static method is meant for doctests, so they can easily construct an example.
> >
> > OUTPUT: an instance of *OutputImageSvg*
> >
> > EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImageSvg
sage: OutputImageSvg.example()
OutputImageSvg container
sage: OutputImageSvg.example().svg
buffer containing 1422 bytes
sage: b'</svg>' in OutputImageSvg.example().svg.get()
True
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputImageSvg
>>> OutputImageSvg.example()
OutputImageSvg container
```

```
>>> OutputImageSvg.example().svg
buffer containing 1422 bytes
>>> b'</svg>' in OutputImageSvg.example().svg.get()
True
```

5.7 Three-Dimensional Graphics Output Types

This module defines the rich output types for 3-d scenes.

class sage.repl.rich_output.output_graphics3d.**OutputSceneCanvas3d**(*canvas3d*)

Bases: *OutputBase*

Canvas3d Scene.

INPUT:

- canvas3d – string/bytes; the canvas3d data

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneCanvas3d
sage: OutputSceneCanvas3d.example()
OutputSceneCanvas3d container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputSceneCanvas3d
>>> OutputSceneCanvas3d.example()
OutputSceneCanvas3d container
```

classmethod example()

Construct a sample Canvas3D output container.

This static method is meant for doctests, so they can easily construct an example.

OUTPUT: an instance of *OutputSceneCanvas3d*

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneCanvas3d
sage: rich_output = OutputSceneCanvas3d.example();  rich_output
OutputSceneCanvas3d container

sage: rich_output.canvas3d
buffer containing 829 bytes
sage: rich_output.canvas3d.get_str()
'[{"vertices":[{"x":1,"y":1,"z":1},...{"x":1,"y":-1,"z":-1}],"faces":[[0,1,2,
↪3]],"color":"008000"}]'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputSceneCanvas3d
>>> rich_output = OutputSceneCanvas3d.example();  rich_output
OutputSceneCanvas3d container

>>> rich_output.canvas3d
```

```
buffer containing 829 bytes
>>> rich_output.canvas3d.get_str()
'[{"vertices":[{"x":1,"y":1,"z":1},...{"x":1,"y":-1,"z":-1}],"faces":[[0,1,2,
→3]],"color":"008000"}]'
```

class sage.repl.rich_output.output_graphics3d.**OutputSceneJmol**(*scene_zip*, *preview_png*)

Bases: *OutputBase*

JMol Scene.

By our (Sage) convention, the actual scene is called SCENE inside the zip archive.

INPUT:

- scene_zip – string/bytes; the jmol scene (a zip archive)

- preview_png – string/bytes; preview as png file

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneJmol
sage: OutputSceneJmol.example()
OutputSceneJmol container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputSceneJmol
>>> OutputSceneJmol.example()
OutputSceneJmol container
```

classmethod example()

Construct a sample Jmol output container.

This static method is meant for doctests, so they can easily construct an example.

OUTPUT: an instance of *OutputSceneJmol*

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneJmol
sage: rich_output = OutputSceneJmol.example();  rich_output
OutputSceneJmol container

sage: rich_output.scene_zip
buffer containing 654 bytes
sage: rich_output.scene_zip.get().startswith(b'PK')
True

sage: rich_output.preview_png
buffer containing 608 bytes
sage: rich_output.preview_png.get().startswith(b'\x89PNG')
True
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputSceneJmol
>>> rich_output = OutputSceneJmol.example();  rich_output
OutputSceneJmol container
```

```
>>> rich_output.scene_zip
buffer containing 654 bytes
>>> rich_output.scene_zip.get().startswith(b'PK')
True

>>> rich_output.preview_png
buffer containing 608 bytes
>>> rich_output.preview_png.get().startswith(b'\x89PNG')
True
```

launch_script_filename()

> Return a launch script suitable to display the scene.
>
> This method saves the scene to disk and creates a launch script. The latter contains an absolute path to the scene file. The launch script is often necessary to make jmol render the 3d scene.
>
> OUTPUT: string; the file name of a suitable launch script
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneJmol
sage: rich_output = OutputSceneJmol.example();  rich_output
OutputSceneJmol container
sage: filename = rich_output.launch_script_filename();  filename
'/.../scene.spt'
sage: with open(filename) as fobj:
....:     print(fobj.read())
set defaultdirectory "/.../scene.spt.zip"
script SCRIPT
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputSceneJmol
>>> rich_output = OutputSceneJmol.example();  rich_output
OutputSceneJmol container
>>> filename = rich_output.launch_script_filename();  filename
'/.../scene.spt'
>>> with open(filename) as fobj:
...     print(fobj.read())
set defaultdirectory "/.../scene.spt.zip"
script SCRIPT
```

class sage.repl.rich_output.output_graphics3d.**OutputSceneThreejs**(*html*)

> Bases: *OutputBase*
>
> Three.js Scene.
>
> INPUT:
>
> - `html` – string/bytes; the Three.js HTML data
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneThreejs
sage: OutputSceneThreejs('<html></html>')
OutputSceneThreejs container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputSceneThreejs
>>> OutputSceneThreejs('<html></html>')
OutputSceneThreejs container
```

class sage.repl.rich_output.output_graphics3d.**OutputSceneWavefront**(*obj*, *mtl*)

Bases: *OutputBase*

Wavefront ∗.*obj* Scene.

The Wavefront format consists of two files, an `.obj` file defining the geometry data (mesh points, normal vectors, …) together with a `.mtl` file defining texture data.

INPUT:

- `obj` – bytes; the Wavefront obj file format describing the mesh shape

- `mtl` – bytes; the Wavefront mtl file format describing textures

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneWavefront
sage: OutputSceneWavefront.example()
OutputSceneWavefront container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputSceneWavefront
>>> OutputSceneWavefront.example()
OutputSceneWavefront container
```

classmethod example()

Construct a sample Canvas3D output container.

This static method is meant for doctests, so they can easily construct an example.

OUTPUT: an instance of *OutputSceneCanvas3d*

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneWavefront
sage: rich_output = OutputSceneWavefront.example();  rich_output
OutputSceneWavefront container

sage: rich_output.obj
buffer containing 227 bytes
sage: rich_output.obj.get_str()
'mtllib scene.mtl\ng obj_1\n...\nf 1 5 6 2\nf 1 4 7 5\nf 6 5 7 8\nf 7 4 3 8\
↪nf 3 2 6 8\n'

sage: rich_output.mtl
buffer containing 80 bytes
sage: rich_output.mtl.get_str()
'newmtl texture177\nKa 0.2 0.2 0.5\nKd 0.4 0.4 1.0\nKs 0.0 0.0 0.0\nillum 1\
↪nNs 1\nd 1\n'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputSceneWavefront
>>> rich_output = OutputSceneWavefront.example();  rich_output
OutputSceneWavefront container

>>> rich_output.obj
buffer containing 227 bytes
>>> rich_output.obj.get_str()
'mtllib scene.mtl\ng obj_1\n...\nf 1 5 6 2\nf 1 4 7 5\nf 6 5 7 8\nf 7 4 3 8\
↪nf 3 2 6 8\n'

>>> rich_output.mtl
buffer containing 80 bytes
>>> rich_output.mtl.get_str()
'newmtl texture177\nKa 0.2 0.2 0.5\nKd 0.4 0.4 1.0\nKs 0.0 0.0 0.0\nillum 1\
↪nNs 1\nd 1\n'
```

mtllib()

Return the `mtllib` filename.

The `mtllib` line in the Wavefront file format (`*.obj`) is the name of the separate texture file.

OUTPUT:

String. The filename under which `mtl` is supposed to be saved.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneWavefront
sage: rich_output = OutputSceneWavefront.example()
sage: rich_output.mtllib()
'scene.mtl'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputSceneWavefront
>>> rich_output = OutputSceneWavefront.example()
>>> rich_output.mtllib()
'scene.mtl'
```

obj_filename()

Return the file name of the `.obj` file.

This method saves the object and texture to separate files in a temporary directory and returns the object file name. This is often used to launch a 3d viewer.

OUTPUT:

String. The file name (absolute path) of the saved obj file.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneWavefront
sage: rich_output = OutputSceneWavefront.example();  rich_output
OutputSceneWavefront container
sage: obj = rich_output.obj_filename();  obj
'/.../scene.obj'
```

```
sage: with open(obj) as fobj:
....:     print(fobj.read())
mtllib scene.mtl
g obj_1
...
f 3 2 6 8

sage: path = os.path.dirname(obj)
sage: mtl = os.path.join(path, 'scene.mtl');  mtl
'/.../scene.mtl'
sage: os.path.exists(mtl)
True
sage: os.path.dirname(obj) == os.path.dirname(mtl)
True
sage: with open(mtl) as fobj:
....:     print(fobj.read())
newmtl texture177
Ka 0.2 0.2 0.5
...
d 1
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputSceneWavefront
>>> rich_output = OutputSceneWavefront.example();  rich_output
OutputSceneWavefront container
>>> obj = rich_output.obj_filename();  obj
'/.../scene.obj'
>>> with open(obj) as fobj:
...     print(fobj.read())
mtllib scene.mtl
g obj_1
...
f 3 2 6 8

>>> path = os.path.dirname(obj)
>>> mtl = os.path.join(path, 'scene.mtl');  mtl
'/.../scene.mtl'
>>> os.path.exists(mtl)
True
>>> os.path.dirname(obj) == os.path.dirname(mtl)
True
>>> with open(mtl) as fobj:
...     print(fobj.read())
newmtl texture177
Ka 0.2 0.2 0.5
...
d 1
```

5.8 Video Output Types

This module defines the rich output types for video formats.

class sage.repl.rich_output.output_video.**OutputVideoAvi**(*video*, *loop=True*)

    Bases: *OutputVideoBase*

    AVI video.

    EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoAvi
sage: OutputVideoAvi.example()
OutputVideoAvi container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputVideoAvi
>>> OutputVideoAvi.example()
OutputVideoAvi container
```

    ext = '.avi'

    mimetype = 'video/x-msvideo'

class sage.repl.rich_output.output_video.**OutputVideoBase**(*video*, *loop=True*)

    Bases: *OutputBase*

    Abstract base class for rich video output.

    INPUT:

        • video – *OutputBuffer*; the video data

        • loop – boolean; whether to repeat the video in an endless loop

    EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoOgg
sage: OutputVideoOgg.example()   # indirect doctest
OutputVideoOgg container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputVideoOgg
>>> OutputVideoOgg.example()   # indirect doctest
OutputVideoOgg container
```

    classmethod example()

        Construct a sample video output container.

        This static method is meant for doctests, so they can easily construct an example. The method is implemented in the abstract *OutputVideoBase* class, but should get invoked on a concrete subclass for which an actual example can exist.

        OUTPUT: an instance of the class on which this method is called

        EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoOgg
sage: OutputVideoOgg.example()
OutputVideoOgg container
sage: OutputVideoOgg.example().video
buffer containing 5612 bytes
sage: OutputVideoOgg.example().ext
'.ogv'
sage: OutputVideoOgg.example().mimetype
'video/ogg'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputVideoOgg
>>> OutputVideoOgg.example()
OutputVideoOgg container
>>> OutputVideoOgg.example().video
buffer containing 5612 bytes
>>> OutputVideoOgg.example().ext
'.ogv'
>>> OutputVideoOgg.example().mimetype
'video/ogg'
```

html_fragment(*url*, *link_attrs=''*)

Construct a HTML fragment for embedding this video.

INPUT:

- `url` – string; the URL where the data of this video can be found

- `link_attrs` – string; can be used to style the fallback link which is presented to the user if the video is not supported

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoOgg
sage: print(OutputVideoOgg.example().html_fragment
....:          ('foo', 'class="bar"').replace('><','>\n<'))
<video autoplay="autoplay" controls="controls" loop="loop">
<source src="foo" type="video/ogg" />
<p>
<a target="_new" href="foo" class="bar">Download video/ogg video</a>
</p>
</video>
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputVideoOgg
>>> print(OutputVideoOgg.example().html_fragment
...         ('foo', 'class="bar"').replace('><','>\n<'))
<video autoplay="autoplay" controls="controls" loop="loop">
<source src="foo" type="video/ogg" />
<p>
<a target="_new" href="foo" class="bar">Download video/ogg video</a>
</p>
</video>
```

class sage.repl.rich_output.output_video.**OutputVideoFlash**(*video*, *loop=True*)

Bases: *OutputVideoBase*

Flash video.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoFlash
sage: OutputVideoFlash.example()
OutputVideoFlash container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputVideoFlash
>>> OutputVideoFlash.example()
OutputVideoFlash container
```

ext = '.flv'

mimetype = 'video/x-flv'

class sage.repl.rich_output.output_video.**OutputVideoMatroska**(*video*, *loop=True*)

Bases: *OutputVideoBase*

Matroska Video.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoMatroska
sage: OutputVideoMatroska.example()
OutputVideoMatroska container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputVideoMatroska
>>> OutputVideoMatroska.example()
OutputVideoMatroska container
```

ext = '.mkv'

mimetype = 'video/x-matroska'

class sage.repl.rich_output.output_video.**OutputVideoMp4**(*video*, *loop=True*)

Bases: *OutputVideoBase*

MPEG 4 video.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoMp4
sage: OutputVideoMp4.example()
OutputVideoMp4 container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputVideoMp4
>>> OutputVideoMp4.example()
OutputVideoMp4 container
```

```
ext = '.mp4'
```

```
mimetype = 'video/mp4'
```

class sage.repl.rich_output.output_video.**OutputVideoOgg**(*video*, *loop=True*)

Bases: *OutputVideoBase*

Ogg video, Ogg Theora in particular.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoOgg
sage: OutputVideoOgg.example()
OutputVideoOgg container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputVideoOgg
>>> OutputVideoOgg.example()
OutputVideoOgg container
```

```
ext = '.ogv'
```

```
mimetype = 'video/ogg'
```

class sage.repl.rich_output.output_video.**OutputVideoQuicktime**(*video*, *loop=True*)

Bases: *OutputVideoBase*

Quicktime video.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoQuicktime
sage: OutputVideoQuicktime.example()
OutputVideoQuicktime container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputVideoQuicktime
>>> OutputVideoQuicktime.example()
OutputVideoQuicktime container
```

```
ext = '.mov'
```

```
mimetype = 'video/quicktime'
```

class sage.repl.rich_output.output_video.**OutputVideoWebM**(*video*, *loop=True*)

Bases: *OutputVideoBase*

WebM video.

The video can be encoded using VP8, VP9 or an even more recent codec.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoWebM
sage: OutputVideoWebM.example()
OutputVideoWebM container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputVideoWebM
>>> OutputVideoWebM.example()
OutputVideoWebM container
```

ext = '.webm'

mimetype = 'video/webm'

class sage.repl.rich_output.output_video.**OutputVideoWmv**(*video*, *loop=True*)

Bases: *OutputVideoBase*

Windows Media Video.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoWmv
sage: OutputVideoWmv.example()
OutputVideoWmv container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_catalog import OutputVideoWmv
>>> OutputVideoWmv.example()
OutputVideoWmv container
```

ext = '.wmv'

mimetype = 'video/x-ms-wmv'

5.9 Catalog of all available output container types.

If you define another output type then you must add it to the imports here.

5.10 Base Class for Backends

The display backends are the commandline, the IPython notebook, the Emacs sage mode, the Sage doctester, …. All of these have different capabilities for what they can display.

To implement a new display backend, you need to subclass *BackendBase*. All backend-specific handling of rich output should be in *displayhook()* and *display_immediately()*. See *BackendSimple* for an absolutely minimal example of a functioning backend.

You declare the types of rich output that your backend can handle in *supported_output()*. There are two ways to then display specific output types in your own backend.

- Directly use one of the existing output containers listed in *sage.repl.rich_output.output_catalog*. That is, test for the rich output type in *display_immediately()* and handle it.

- Subclass the rich output container to attach your backend-specific functionality. Then *display_immediately()* will receive instances of your subclass. See *BackendTest* for an example of how this is done.

You can also mix both ways of implementing different rich output types.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendSimple
sage: backend = BackendSimple()
sage: plain_text = backend.plain_text_formatter(list(range(10)));  plain_text
OutputPlainText container
sage: backend.displayhook(plain_text, plain_text)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_base import BackendSimple
>>> backend = BackendSimple()
>>> plain_text = backend.plain_text_formatter(list(range(Integer(10))));  plain_text
OutputPlainText container
>>> backend.displayhook(plain_text, plain_text)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

class sage.repl.rich_output.backend_base.**BackendBase**

    Bases: SageObject

    ascii_art_formatter(*obj*, ***kwds*)

        Hook to override how ascii art is being formatted.

        INPUT:

            • obj – anything

            • **kwds – optional keyword arguments to control the formatting. Supported are:

                – concatenate – boolean (default: False); if True, the argument obj must be iterable and its entries will be concatenated. There is a single whitespace between entries.

        OUTPUT:

        Instance of *OutputAsciiArt* containing the ascii art string representation of the object.

        EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: out = backend.ascii_art_formatter(list(range(30)))
sage: out
OutputAsciiArt container
sage: out.ascii_art
buffer containing 114 bytes
sage: print(out.ascii_art.get_str())
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,␣
↪21,

 22, 23, 24, 25, 26, 27, 28, 29 ]
sage: backend.ascii_art_formatter([1,2,3], concatenate=False).ascii_art.get_
↪str()
'[ 1, 2, 3 ]'
sage: backend.ascii_art_formatter([1,2,3], concatenate=True).ascii_art.get_
↪str()
'1 2 3'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_base import BackendBase
>>> backend = BackendBase()
>>> out = backend.ascii_art_formatter(list(range(Integer(30))))
>>> out
OutputAsciiArt container
>>> out.ascii_art
buffer containing 114 bytes
>>> print(out.ascii_art.get_str())
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,␣
↪21,
<BLANKLINE>
 22, 23, 24, 25, 26, 27, 28, 29 ]
>>> backend.ascii_art_formatter([Integer(1),Integer(2),Integer(3)],␣
↪concatenate=False).ascii_art.get_str()
'[ 1, 2, 3 ]'
>>> backend.ascii_art_formatter([Integer(1),Integer(2),Integer(3)],␣
↪concatenate=True).ascii_art.get_str()
'1 2 3'
```

default_preferences()

> Return the backend's display preferences.
>
> Override this method to change the default preferences when using your backend.
>
> OUTPUT:
>
> Instance of *DisplayPreferences*.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.default_preferences()
Display preferences:
* align_latex is not specified
* graphics is not specified
* supplemental_plot is not specified
* text is not specified
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_base import BackendBase
>>> backend = BackendBase()
>>> backend.default_preferences()
Display preferences:
* align_latex is not specified
* graphics is not specified
* supplemental_plot is not specified
* text is not specified
```

display_immediately(*plain_text*, *rich_output*)

> Show output without going back to the command line prompt.
>
> This method is similar to the rich output *displayhook()*, except that it can be invoked at any time. Typically, it ends up being called by `sage.plot.graphics.Graphics.show()`.

Derived classes must implement this method.

INPUT:

Same as *displayhook()*.

OUTPUT:

This method may return something so you can implement *displayhook()* by calling this method. However, when called by the display manager any potential return value is discarded: There is no way to return anything without returning to the command prompt.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_basic import OutputPlainText
sage: plain_text = OutputPlainText.example()
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.display_immediately(plain_text, plain_text)
Traceback (most recent call last):
...
NotImplementedError: derived classes must implement this method
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_basic import OutputPlainText
>>> plain_text = OutputPlainText.example()
>>> from sage.repl.rich_output.backend_base import BackendBase
>>> backend = BackendBase()
>>> backend.display_immediately(plain_text, plain_text)
Traceback (most recent call last):
...
NotImplementedError: derived classes must implement this method
```

displayhook(*plain_text*, *rich_output*)

Backend implementation of the displayhook.

The value of the last statement on a REPL input line or notebook cell are usually handed to the Python displayhook and shown on screen. By overriding this method you define how your backend handles output. The difference to the usual displayhook is that Sage already converted the value to the most suitable rich output container.

Derived classes must implement this method.

INPUT:

- `plain_text` – instance of *OutputPlainText*. The plain text version of the output.

- `rich_output` – instance of an output container class (subclass of *OutputBase*). Guaranteed to be one of the output containers returned from *supported_output()*, possibly the same as `plain_text`.

OUTPUT:

This method may return something, which is then returned from the display manager's *displayhook()* method.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_basic import OutputPlainText
sage: plain_text = OutputPlainText.example()
```

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.displayhook(plain_text, plain_text)
Traceback (most recent call last):
...
NotImplementedError: derived classes must implement this method
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_basic import OutputPlainText
>>> plain_text = OutputPlainText.example()
>>> from sage.repl.rich_output.backend_base import BackendBase
>>> backend = BackendBase()
>>> backend.displayhook(plain_text, plain_text)
Traceback (most recent call last):
...
NotImplementedError: derived classes must implement this method
```

get_display_manager()

Return the display manager singleton.

This is a convenience method to access the display manager singleton.

OUTPUT:

The unique *DisplayManager* instance.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.get_display_manager()
The Sage display manager using the doctest backend
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_base import BackendBase
>>> backend = BackendBase()
>>> backend.get_display_manager()
The Sage display manager using the doctest backend
```

install(***kwds*)

Hook that will be called once before the backend is used for the first time.

The default implementation does nothing.

INPUT:

- kwds – optional keyword arguments that are passed through by the *switch_backend()* method.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.install()
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_base import BackendBase
>>> backend = BackendBase()
>>> backend.install()
```

is_in_terminal()

> Test whether the UI is meant to run in a terminal.
>
> See *sage.repl.rich_output.display_manager.DisplayManager.is_in_terminal()* for details.
>
> OUTPUT: defaults to `False`
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.is_in_terminal()
False
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_base import BackendBase
>>> backend = BackendBase()
>>> backend.is_in_terminal()
False
```

latex_formatter(*obj*, ***kwds*)

> Hook to override how latex is being formatted.
>
> INPUT:
>
> - `obj` – anything
>
> - `**kwds` – optional keyword arguments to control the formatting. Supported are:
>
>   - `concatenate` – boolean (default: `False`); if `True`, the argument `obj` must be iterable and its entries will be concatenated. There is a single whitespace between entries.
>
> OUTPUT:
>
> Instance of `OutputHtml` containing the latex string representation of the object.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: out = backend.latex_formatter(1/2)
sage: out
OutputHtml container
sage: out.html
buffer containing 42 bytes
sage: out.html.get_str()
'<html>\\(\\displaystyle \\frac{1}{2}\\)</html>'

sage: # needs sage.symbolic
sage: out = backend.latex_formatter([1/2, x, 3/4, ZZ], concatenate=False)
sage: out.html.get_str()
```

```
'<html>\\(\\displaystyle \\newcommand{\\Bold}[1]{\\mathbf{#1}}\\left[\\frac{1}
→{2}, x, \\frac{3}{4}, \\Bold{Z}\\right]\\)</html>'
sage: out = backend.latex_formatter([1/2, x, 3/4, ZZ], concatenate=True)
sage: out.html.get_str()
'<html>\\(\\displaystyle \\newcommand{\\Bold}[1]{\\mathbf{#1}}\\frac{1}{2} x \
→\\frac{3}{4} \\Bold{Z}\\)</html>'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_base import BackendBase
>>> backend = BackendBase()
>>> out = backend.latex_formatter(Integer(1)/Integer(2))
>>> out
OutputHtml container
>>> out.html
buffer containing 42 bytes
>>> out.html.get_str()
'<html>\\(\\displaystyle \\frac{1}{2}\\)</html>'

>>> # needs sage.symbolic
>>> out = backend.latex_formatter([Integer(1)/Integer(2), x, Integer(3)/
→Integer(4), ZZ], concatenate=False)
>>> out.html.get_str()
'<html>\\(\\displaystyle \\newcommand{\\Bold}[1]{\\mathbf{#1}}\\left[\\frac{1}
→{2}, x, \\frac{3}{4}, \\Bold{Z}\\right]\\)</html>'
>>> out = backend.latex_formatter([Integer(1)/Integer(2), x, Integer(3)/
→Integer(4), ZZ], concatenate=True)
>>> out.html.get_str()
'<html>\\(\\displaystyle \\newcommand{\\Bold}[1]{\\mathbf{#1}}\\frac{1}{2} x \
→\\frac{3}{4} \\Bold{Z}\\)</html>'
```

max_width()

Return the number of characters that fit into one output line.

OUTPUT: integer

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.max_width()
79
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_base import BackendBase
>>> backend = BackendBase()
>>> backend.max_width()
79
```

newline()

Return the newline string.

OUTPUT: string for starting a new line of output

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.newline()
'\n'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_base import BackendBase
>>> backend = BackendBase()
>>> backend.newline()
'\n'
```

plain_text_formatter(*obj*, ***kwds*)

Hook to override how plain text is being formatted.

If the object does not have a _rich_repr_ method, or if it does not return a rich output object (*Output-Base*), then this method is used to generate plain text output.

INPUT:

- obj – anything

- **kwds – optional keyword arguments to control the formatting. Supported are:

    - concatenate – boolean (default: False); if True, the argument obj must be iterable and its entries will be concatenated. There is a single whitespace between entries.

OUTPUT:

Instance of *OutputPlainText* containing the string representation of the object.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: out = backend.plain_text_formatter(list(range(30)))
sage: out
OutputPlainText container
sage: out.text
buffer containing 139 bytes
sage: out.text.get_str()
'[0,\n 1,\n 2,\n 3,\n 4,\n 5,\n 6,\n 7,\n 8,\n 9,\n
10,\n 11,\n 12,\n 13,\n 14,\n 15,\n 16,\n 17,\n 18,\n
19,\n 20,\n 21,\n 22,\n 23,\n 24,\n 25,\n 26,\n 27,\n
28,\n 29]'

sage: out = backend.plain_text_formatter(list(range(20)), concatenate=True)
sage: out.text.get_str()
'0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_base import BackendBase
>>> backend = BackendBase()
>>> out = backend.plain_text_formatter(list(range(Integer(30))))
>>> out
OutputPlainText container
>>> out.text
```

```
buffer containing 139 bytes
>>> out.text.get_str()
'[0,\n 1,\n 2,\n 3,\n 4,\n 5,\n 6,\n 7,\n 8,\n 9,\n
10,\n 11,\n 12,\n 13,\n 14,\n 15,\n 16,\n 17,\n 18,\n
19,\n 20,\n 21,\n 22,\n 23,\n 24,\n 25,\n 26,\n 27,\n
28,\n 29]'

>>> out = backend.plain_text_formatter(list(range(Integer(20))),
↪concatenate=True)
>>> out.text.get_str()
'0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19'
```

set_underscore_variable(*obj*)

> Set the _ builtin variable.
>
> By default, this sets the special _ variable. Backends that organize the history differently (e.g. IPython) can override this method.
>
> INPUT:
>
> > • obj – result of the most recent evaluation
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.set_underscore_variable(123)
sage: _
123

sage: 'foo'
'foo'
sage: _        # indirect doctest
'foo'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_base import BackendBase
>>> backend = BackendBase()
>>> backend.set_underscore_variable(Integer(123))
>>> _
123

>>> 'foo'
'foo'
>>> _        # indirect doctest
'foo'
```

supported_output()

> Return the outputs that are supported by the backend.
>
> Subclasses must implement this method.
>
> OUTPUT:
>
> Iterable of output container classes, that is, subclass of *OutputBase*). May be a list/tuple/set/frozenset. The order is ignored. Only used internally by the display manager.

You may return backend-specific subclasses of existing output containers. This allows you to attach backend-specific functionality to the output container.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.supported_output()
Traceback (most recent call last):
...
NotImplementedError: derived classes must implement this method
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_base import BackendBase
>>> backend = BackendBase()
>>> backend.supported_output()
Traceback (most recent call last):
...
NotImplementedError: derived classes must implement this method
```

unicode_art_formatter(*obj*, ***kwds*)

Hook to override how unicode art is being formatted.

INPUT:

- `obj` – anything

- `**kwds` – optional keyword arguments to control the formatting. Supported are:

    - `concatenate` – boolean (default: `False`); if `True`, the argument `obj` must be iterable and its entries will be concatenated. There is a single whitespace between entries.

OUTPUT:

Instance of *OutputUnicodeArt* containing the unicode art string representation of the object.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: out = backend.unicode_art_formatter(list(range(30)))
sage: out
OutputUnicodeArt container
sage: out.unicode_art
buffer containing 114 bytes
sage: print(out.unicode_art.get_str())
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,␣
↪21,

22, 23, 24, 25, 26, 27, 28, 29 ]

sage: backend.unicode_art_formatter([1,2,3], concatenate=False).unicode_art.
↪get_str()
'[ 1, 2, 3 ]'
sage: backend.unicode_art_formatter([1,2,3], concatenate=True).unicode_art.
↪get_str()
'1 2 3'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_base import BackendBase
>>> backend = BackendBase()
>>> out = backend.unicode_art_formatter(list(range(Integer(30))))
>>> out
OutputUnicodeArt container
>>> out.unicode_art
buffer containing 114 bytes
>>> print(out.unicode_art.get_str())
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,␣
↪21,
<BLANKLINE>
22, 23, 24, 25, 26, 27, 28, 29 ]

>>> backend.unicode_art_formatter([Integer(1),Integer(2),Integer(3)],␣
↪concatenate=False).unicode_art.get_str()
'[ 1, 2, 3 ]'
>>> backend.unicode_art_formatter([Integer(1),Integer(2),Integer(3)],␣
↪concatenate=True).unicode_art.get_str()
'1 2 3'
```

uninstall()

> Hook that will be called once right before the backend is removed.
>
> The default implementation does nothing.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.uninstall()
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_base import BackendBase
>>> backend = BackendBase()
>>> backend.uninstall()
```

class sage.repl.rich_output.backend_base.**BackendSimple**

> Bases: *BackendBase*
>
> Simple Backend.
>
> This backend only supports plain text.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendSimple
sage: BackendSimple()
simple
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_base import BackendSimple
>>> BackendSimple()
simple
```

display_immediately(*plain_text*, *rich_output*)

> Show output without going back to the command line prompt.
>
> INPUT:
>
> Same as *displayhook()*.
>
> OUTPUT: this backend returns nothing, it just prints to stdout
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.output_basic import OutputPlainText
sage: plain_text = OutputPlainText.example()
sage: from sage.repl.rich_output.backend_base import BackendSimple
sage: backend = BackendSimple()
sage: backend.display_immediately(plain_text, plain_text)
Example plain text output
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_basic import OutputPlainText
>>> plain_text = OutputPlainText.example()
>>> from sage.repl.rich_output.backend_base import BackendSimple
>>> backend = BackendSimple()
>>> backend.display_immediately(plain_text, plain_text)
Example plain text output
```

supported_output()

> Return the outputs that are supported by the backend.
>
> OUTPUT:
>
> Iterable of output container classes, that is, subclass of *OutputBase*). This backend only supports the plain text output container.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendSimple
sage: backend = BackendSimple()
sage: backend.supported_output()
{<class 'sage.repl.rich_output.output_basic.OutputPlainText'>}
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_base import BackendSimple
>>> backend = BackendSimple()
>>> backend.supported_output()
{<class 'sage.repl.rich_output.output_basic.OutputPlainText'>}
```

5.11 Test Backend

This backend is only for doctesting purposes.

EXAMPLES:

We switch to the test backend for the remainder of this file:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: from sage.repl.rich_output.test_backend import BackendTest, TestObject
sage: doctest_backend = dm.switch_backend(BackendTest())
sage: dm
The Sage display manager using the test backend

sage: dm._output_promotions
{<class 'sage.repl.rich_output.output_basic.OutputPlainText'>:
 <class 'sage.repl.rich_output.test_backend.TestOutputPlainText'>}
sage: dm.displayhook(1/2)
1/2 [TestOutputPlainText]
TestOutputPlainText container

sage: test = TestObject()
sage: test
called the _repr_ method
sage: dm.displayhook(test)
called the _rich_repr_ method [TestOutputPlainText]
TestOutputPlainText container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output import get_display_manager
>>> dm = get_display_manager()
>>> from sage.repl.rich_output.test_backend import BackendTest, TestObject
>>> doctest_backend = dm.switch_backend(BackendTest())
>>> dm
The Sage display manager using the test backend

>>> dm._output_promotions
{<class 'sage.repl.rich_output.output_basic.OutputPlainText'>:
 <class 'sage.repl.rich_output.test_backend.TestOutputPlainText'>}
>>> dm.displayhook(Integer(1)/Integer(2))
1/2 [TestOutputPlainText]
TestOutputPlainText container

>>> test = TestObject()
>>> test
called the _repr_ method
>>> dm.displayhook(test)
called the _rich_repr_ method [TestOutputPlainText]
TestOutputPlainText container
```

class sage.repl.rich_output.test_backend.**BackendTest**

>   Bases: *BackendBase*

>   **display_immediately**(*plain_text*, *rich_output*)

> >   Show output without going back to the command line prompt.

> >   INPUT:

> >   Same as `displayhook()`.

> >   OUTPUT:

This method returns the rich output for doctesting convenience. The actual display framework ignores the return value.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_basic import OutputPlainText
sage: plain_text = OutputPlainText.example()
sage: from sage.repl.rich_output.test_backend import BackendTest
sage: backend = BackendTest()
sage: backend.display_immediately(plain_text, plain_text)
Example plain text output
OutputPlainText container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_basic import OutputPlainText
>>> plain_text = OutputPlainText.example()
>>> from sage.repl.rich_output.test_backend import BackendTest
>>> backend = BackendTest()
>>> backend.display_immediately(plain_text, plain_text)
Example plain text output
OutputPlainText container
```

supported_output()

Return the outputs that are supported by the backend.

OUTPUT:

Iterable of output container classes. Only the `TestOutputPlainText` output container is supported by the test backend.

EXAMPLES:

```
sage: display_manager = sage.repl.rich_output.get_display_manager()
sage: backend = display_manager._backend
sage: list(backend.supported_output())
[<class 'sage.repl.rich_output.test_backend.TestOutputPlainText'>]
```

```
>>> from sage.all import *
>>> display_manager = sage.repl.rich_output.get_display_manager()
>>> backend = display_manager._backend
>>> list(backend.supported_output())
[<class 'sage.repl.rich_output.test_backend.TestOutputPlainText'>]
```

The output of this method is used by the display manager to set up the actual supported outputs. Compare:

```
sage: list(display_manager.supported_output())
[<class 'sage.repl.rich_output.output_basic.OutputPlainText'>]
```

```
>>> from sage.all import *
>>> list(display_manager.supported_output())
[<class 'sage.repl.rich_output.output_basic.OutputPlainText'>]
```

class sage.repl.rich_output.test_backend.**TestObject**

Bases: `SageObject`

Test object with both _repr_() and _rich_repr_()

---

class sage.repl.rich_output.test_backend.**TestOutputPlainText**(**args*, ***kwds*)

> Bases: *OutputPlainText*

Backend-specific subclass of the plain text output container.

Backends must not influence how the display system constructs output containers, they can only control how the output container is displayed. In particular, we cannot override the constructor (only the *OutputPlainText* constructor is used).

EXAMPLES:

```
sage: from sage.repl.rich_output.test_backend import TestOutputPlainText
sage: TestOutputPlainText()
Traceback (most recent call last):
...
AssertionError: cannot override constructor
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.test_backend import TestOutputPlainText
>>> TestOutputPlainText()
Traceback (most recent call last):
...
AssertionError: cannot override constructor
```

> **print_to_stdout**()
>
> > Write the data to stdout.
> >
> > This is just a convenience method to help with debugging.
> >
> > EXAMPLES:
> >
> > ```
> > sage: from sage.repl.rich_output import get_display_manager
> > sage: dm = get_display_manager()
> > sage: test_output = dm.displayhook(123)
> > 123 [TestOutputPlainText]
> > sage: type(test_output)
> > <class 'sage.repl.rich_output.test_backend.TestOutputPlainText'>
> > sage: test_output.print_to_stdout()
> > 123 [TestOutputPlainText]
> > ```
> >
> > ```
> > >>> from sage.all import *
> > >>> from sage.repl.rich_output import get_display_manager
> > >>> dm = get_display_manager()
> > >>> test_output = dm.displayhook(Integer(123))
> > 123 [TestOutputPlainText]
> > >>> type(test_output)
> > <class 'sage.repl.rich_output.test_backend.TestOutputPlainText'>
> > >>> test_output.print_to_stdout()
> > 123 [TestOutputPlainText]
> > ```

5.12 The backend used for doctests

This backend is active during doctests. It should mimic the behavior of the IPython command line as close as possible. Without actually launching image viewers, of course.

EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: get_display_manager()
The Sage display manager using the doctest backend
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output import get_display_manager
>>> get_display_manager()
The Sage display manager using the doctest backend
```

class sage.repl.rich_output.backend_doctest.**BackendDoctest**

Bases: *BackendBase*

default_preferences()

Return the backend's display preferences.

Matches the IPython command line display preferences to keep the differences between that and the doctests to a minimum.

OUTPUT:

Instance of *DisplayPreferences*.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import␣
 ↪BackendIPythonCommandline
sage: backend = BackendIPythonCommandline()
sage: backend.default_preferences()
Display preferences:
* align_latex is not specified
* graphics is not specified
* supplemental_plot = never
* text is not specified
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_ipython import␣
 ↪BackendIPythonCommandline
>>> backend = BackendIPythonCommandline()
>>> backend.default_preferences()
Display preferences:
* align_latex is not specified
* graphics is not specified
* supplemental_plot = never
* text is not specified
```

display_immediately(*plain_text*, *rich_output*)

Display object immediately.

INPUT:

Same as *displayhook()*.

EXAMPLES:

The following example does not call the displayhook. More precisely, the `show()` method returns `None` which is ignored by the displayhook. When running the example on a Sage display backend capable of displaying graphics outside of the displayhook, the plot is still shown. Nothing is shown during doctests:

```
sage: plt = plot(sin)                                                    #␣
↪needs sage.plot sage.symbolic
sage: plt                                                                #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
sage: plt.show()                                                         #␣
↪needs sage.plot sage.symbolic

sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: dm.display_immediately(plt)    # indirect doctest                  #␣
↪needs sage.plot sage.symbolic
```

```
>>> from sage.all import *
>>> plt = plot(sin)                                                      #␣
↪needs sage.plot sage.symbolic
>>> plt                                                                  #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
>>> plt.show()                                                          #␣
↪needs sage.plot sage.symbolic

>>> from sage.repl.rich_output import get_display_manager
>>> dm = get_display_manager()
>>> dm.display_immediately(plt)    # indirect doctest                    #␣
↪needs sage.plot sage.symbolic
```

displayhook(*plain_text*, *rich_output*)

Display object from displayhook.

INPUT:

- `plain_text` – instance of `OutputPlainText`. The plain text version of the output.

- `rich_output` – instance of an output container class (subclass of `OutputBase`). Guaranteed to be one of the output containers returned from `supported_output()`, possibly the same as `plain_text`.

EXAMPLES:

This ends up calling the displayhook:

```
sage: plt = plot(sin)                                                    #␣
↪needs sage.plot sage.symbolic
sage: plt                                                                #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
sage: plt.show()                                                         #␣
↪needs sage.plot sage.symbolic

sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: dm.displayhook(plt)          # indirect doctest                    #␣
```

(continues on next page)

```
 →needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> plt = plot(sin)                                                    #␣
 →needs sage.plot sage.symbolic
>>> plt                                                                #␣
 →needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
>>> plt.show()                                                         #␣
 →needs sage.plot sage.symbolic

>>> from sage.repl.rich_output import get_display_manager
>>> dm = get_display_manager()
>>> dm.displayhook(plt)          # indirect doctest                    #␣
 →needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
```

install(***kwds*)

> Switch to the doctest backend.
>
> This method is being called from within *switch_backend()*. You should never call it by hand.
>
> INPUT:
>
> - `**kwds` – none of the optional keyword arguments are used in the doctest backend
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.backend_doctest import BackendDoctest
sage: backend = BackendDoctest()
sage: backend.install()
sage: backend.uninstall()
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_doctest import BackendDoctest
>>> backend = BackendDoctest()
>>> backend.install()
>>> backend.uninstall()
```

supported_output()

> Return the supported output types.
>
> OUTPUT:
>
> Set of subclasses of *OutputBase*, the supported output container types.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.backend_doctest import BackendDoctest
sage: from sage.repl.rich_output.output_catalog import *
sage: backend = BackendDoctest()
sage: OutputPlainText in backend.supported_output()
True
```

```
sage: OutputSceneJmol in backend.supported_output()
True
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_doctest import BackendDoctest
>>> from sage.repl.rich_output.output_catalog import *
>>> backend = BackendDoctest()
>>> OutputPlainText in backend.supported_output()
True
>>> OutputSceneJmol in backend.supported_output()
True
```

uninstall()

> Switch away from the doctest backend.
>
> This method is being called from within *switch_backend()*. You should never call it by hand.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.backend_doctest import BackendDoctest
sage: backend = BackendDoctest()
sage: backend.install()
sage: backend.uninstall()
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_doctest import BackendDoctest
>>> backend = BackendDoctest()
>>> backend.install()
>>> backend.uninstall()
```

validate(*rich_output*)

> Perform checks on rich_output.
>
> INPUT:
>
> - rich_output – instance of a subclass of *OutputBase*
>
> OUTPUT: an assertion is triggered if rich_output is invalid
>
> EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: invalid = dm.types.OutputImagePng('invalid')
sage: backend = dm._backend;  backend
doctest
sage: backend.validate(invalid)
Traceback (most recent call last):
...
AssertionError
sage: backend.validate(dm.types.OutputPlainText.example())
sage: backend.validate(dm.types.OutputAsciiArt.example())
sage: backend.validate(dm.types.OutputLatex.example())
sage: backend.validate(dm.types.OutputImagePng.example())
```

```
sage: backend.validate(dm.types.OutputImageGif.example())
sage: backend.validate(dm.types.OutputImageJpg.example())
sage: backend.validate(dm.types.OutputImageSvg.example())
sage: backend.validate(dm.types.OutputImagePdf.example())
sage: backend.validate(dm.types.OutputImageDvi.example())
sage: backend.validate(dm.types.OutputSceneJmol.example())
sage: backend.validate(dm.types.OutputSceneWavefront.example())
sage: backend.validate(dm.types.OutputSceneCanvas3d.example())
sage: backend.validate(dm.types.OutputVideoOgg.example())
sage: backend.validate(dm.types.OutputVideoWebM.example())
sage: backend.validate(dm.types.OutputVideoMp4.example())
sage: backend.validate(dm.types.OutputVideoFlash.example())
sage: backend.validate(dm.types.OutputVideoMatroska.example())
sage: backend.validate(dm.types.OutputVideoAvi.example())
sage: backend.validate(dm.types.OutputVideoWmv.example())
sage: backend.validate(dm.types.OutputVideoQuicktime.example())
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output import get_display_manager
>>> dm = get_display_manager()
>>> invalid = dm.types.OutputImagePng('invalid')
>>> backend = dm._backend;  backend
doctest
>>> backend.validate(invalid)
Traceback (most recent call last):
...
AssertionError
>>> backend.validate(dm.types.OutputPlainText.example())
>>> backend.validate(dm.types.OutputAsciiArt.example())
>>> backend.validate(dm.types.OutputLatex.example())
>>> backend.validate(dm.types.OutputImagePng.example())
>>> backend.validate(dm.types.OutputImageGif.example())
>>> backend.validate(dm.types.OutputImageJpg.example())
>>> backend.validate(dm.types.OutputImageSvg.example())
>>> backend.validate(dm.types.OutputImagePdf.example())
>>> backend.validate(dm.types.OutputImageDvi.example())
>>> backend.validate(dm.types.OutputSceneJmol.example())
>>> backend.validate(dm.types.OutputSceneWavefront.example())
>>> backend.validate(dm.types.OutputSceneCanvas3d.example())
>>> backend.validate(dm.types.OutputVideoOgg.example())
>>> backend.validate(dm.types.OutputVideoWebM.example())
>>> backend.validate(dm.types.OutputVideoMp4.example())
>>> backend.validate(dm.types.OutputVideoFlash.example())
>>> backend.validate(dm.types.OutputVideoMatroska.example())
>>> backend.validate(dm.types.OutputVideoAvi.example())
>>> backend.validate(dm.types.OutputVideoWmv.example())
>>> backend.validate(dm.types.OutputVideoQuicktime.example())
```

5.13 IPython Backend for the Sage Rich Output System

This module defines the IPython backends for `sage.repl.rich_output`.

class sage.repl.rich_output.backend_ipython.**BackendIPython**

Bases: *BackendBase*

Common base for the IPython UIs.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import BackendIPython
sage: BackendIPython()._repr_()
Traceback (most recent call last):
...
NotImplementedError: derived classes must implement this method
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_ipython import BackendIPython
>>> BackendIPython()._repr_()
Traceback (most recent call last):
...
NotImplementedError: derived classes must implement this method
```

display_immediately(*plain_text*, *rich_output*)

Show output immediately.

This method is similar to the rich output `displayhook()`, except that it can be invoked at any time.

INPUT:

Same as `displayhook()`.

OUTPUT: this method does not return anything

EXAMPLES:

```
sage: from sage.repl.rich_output.output_basic import OutputPlainText
sage: plain_text = OutputPlainText.example()
sage: from sage.repl.rich_output.backend_ipython import BackendIPythonNotebook
sage: backend = BackendIPythonNotebook()
sage: _ = backend.display_immediately(plain_text, plain_text)
Example plain text output
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_basic import OutputPlainText
>>> plain_text = OutputPlainText.example()
>>> from sage.repl.rich_output.backend_ipython import BackendIPythonNotebook
>>> backend = BackendIPythonNotebook()
>>> _ = backend.display_immediately(plain_text, plain_text)
Example plain text output
```

install(***kwds*)

Switch the Sage rich output to the IPython backend.

INPUT:

- `shell` – keyword argument; the IPython shell

No tests since switching away from the doctest rich output backend will break the doctests.

EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: from sage.repl.rich_output.backend_ipython import BackendIPython
sage: backend = BackendIPython()
sage: shell = get_test_shell()
sage: backend.install(shell=shell)
sage: shell.run_cell('1+1')
2
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell
>>> from sage.repl.rich_output.backend_ipython import BackendIPython
>>> backend = BackendIPython()
>>> shell = get_test_shell()
>>> backend.install(shell=shell)
>>> shell.run_cell('1+1')
2
```

set_underscore_variable(*obj*)

> Set the _ builtin variable.
>
> Since IPython handles the history itself, this does nothing.
>
> INPUT:
>
> - obj – anything
>
> EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: from sage.repl.rich_output.backend_ipython import BackendIPython
sage: backend = BackendIPython()
sage: backend.set_underscore_variable(123)
sage: _
0
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell
>>> from sage.repl.rich_output.backend_ipython import BackendIPython
>>> backend = BackendIPython()
>>> backend.set_underscore_variable(Integer(123))
>>> _
0
```

class sage.repl.rich_output.backend_ipython.**BackendIPythonCommandline**

> Bases: *BackendIPython*
>
> Backend for the IPython Command Line.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import BackendIPythonCommandline
sage: BackendIPythonCommandline()
IPython command line
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_ipython import BackendIPythonCommandline
>>> BackendIPythonCommandline()
IPython command line
```

default_preferences()

> Return the backend's display preferences.
>
> The default for the commandline is to not plot graphs since the launching of an external viewer is considered too disruptive.
>
> OUTPUT:
>
> Instance of *DisplayPreferences*.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import␣
↪BackendIPythonCommandline
sage: backend = BackendIPythonCommandline()
sage: backend.default_preferences()
Display preferences:
* align_latex is not specified
* graphics is not specified
* supplemental_plot = never
* text is not specified
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_ipython import␣
↪BackendIPythonCommandline
>>> backend = BackendIPythonCommandline()
>>> backend.default_preferences()
Display preferences:
* align_latex is not specified
* graphics is not specified
* supplemental_plot = never
* text is not specified
```

display_immediately(*plain_text*, *rich_output*)

> Show output without going back to the command line prompt.
>
> This method is similar to the rich output *displayhook()*, except that it can be invoked at any time. On the Sage command line it launches viewers just like *displayhook()*.
>
> INPUT:
>
> Same as *displayhook()*.
>
> OUTPUT: this method does not return anything
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.output_basic import OutputPlainText
sage: plain_text = OutputPlainText.example()
sage: from sage.repl.rich_output.backend_ipython import␣
↪BackendIPythonCommandline
sage: backend = BackendIPythonCommandline()
```

```
sage: backend.display_immediately(plain_text, plain_text)
Example plain text output
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_basic import OutputPlainText
>>> plain_text = OutputPlainText.example()
>>> from sage.repl.rich_output.backend_ipython import␣
↪BackendIPythonCommandline
>>> backend = BackendIPythonCommandline()
>>> backend.display_immediately(plain_text, plain_text)
Example plain text output
```

displayhook(*plain_text*, *rich_output*)

Backend implementation of the displayhook.

INPUT:

- plain_text – instance of *OutputPlainText*. The plain text version of the output.
- rich_output – instance of an output container class (subclass of *OutputBase*). Guaranteed to be one of the output containers returned from *supported_output()*, possibly the same as plain_text.

OUTPUT:

The IPython commandline display hook returns the IPython display data, a pair of dictionaries. The first dictionary contains mime types as keys and the respective output as value. The second dictionary is metadata.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_basic import OutputPlainText
sage: plain_text = OutputPlainText.example()
sage: from sage.repl.rich_output.backend_ipython import␣
↪BackendIPythonCommandline
sage: backend = BackendIPythonCommandline()
sage: backend.displayhook(plain_text, plain_text)
({'text/plain': 'Example plain text output'}, {})
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_basic import OutputPlainText
>>> plain_text = OutputPlainText.example()
>>> from sage.repl.rich_output.backend_ipython import␣
↪BackendIPythonCommandline
>>> backend = BackendIPythonCommandline()
>>> backend.displayhook(plain_text, plain_text)
({'text/plain': 'Example plain text output'}, {})
```

is_in_terminal()

Test whether the UI is meant to run in a terminal.

See *sage.repl.rich_output.display_manager.DisplayManager.is_in_terminal()* for details.

OUTPUT: True for the IPython commandline

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import␣
↪BackendIPythonCommandline
sage: backend = BackendIPythonCommandline()
sage: backend.is_in_terminal()
True
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_ipython import␣
↪BackendIPythonCommandline
>>> backend = BackendIPythonCommandline()
>>> backend.is_in_terminal()
True
```

launch_jmol(*output_jmol*, *plain_text*)

Launch the stand-alone jmol viewer.

INPUT:

- output_jmol – *OutputSceneJmol*; the scene to launch Jmol with

- plain_text – string; the plain text representation

OUTPUT:

string; human-readable message indicating that the viewer was launched.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import␣
↪BackendIPythonCommandline
sage: backend = BackendIPythonCommandline()
sage: from sage.repl.rich_output.output_graphics3d import OutputSceneJmol
sage: backend.launch_jmol(OutputSceneJmol.example(), 'Graphics3d object')   #␣
↪needs sage.plot
'Launched jmol viewer for Graphics3d object'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_ipython import␣
↪BackendIPythonCommandline
>>> backend = BackendIPythonCommandline()
>>> from sage.repl.rich_output.output_graphics3d import OutputSceneJmol
>>> backend.launch_jmol(OutputSceneJmol.example(), 'Graphics3d object')   #␣
↪needs sage.plot
'Launched jmol viewer for Graphics3d object'
```

launch_viewer(*image_file*, *plain_text*)

Launch external viewer for the graphics file.

INPUT:

- image_file – string; file name of the image file

- plain_text – string; the plain text representation of the image file

OUTPUT:

String. Human-readable message indicating whether the viewer was launched successfully.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import␣
→BackendIPythonCommandline
sage: backend = BackendIPythonCommandline()
sage: backend.launch_viewer('/path/to/foo.bar', 'Graphics object')
'Launched bar viewer for Graphics object'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_ipython import␣
→BackendIPythonCommandline
>>> backend = BackendIPythonCommandline()
>>> backend.launch_viewer('/path/to/foo.bar', 'Graphics object')
'Launched bar viewer for Graphics object'
```

supported_output()

> Return the outputs that are supported by the IPython commandline backend.
>
> OUTPUT:
>
> Iterable of output container classes, that is, subclass of `OutputBase`). The order is ignored.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import␣
→BackendIPythonCommandline
sage: backend = BackendIPythonCommandline()
sage: supp = backend.supported_output();  supp      # random output
set([<class 'sage.repl.rich_output.output_graphics.OutputImageGif'>,
     ...,
     <class 'sage.repl.rich_output.output_graphics.OutputImagePng'>])
sage: from sage.repl.rich_output.output_basic import OutputLatex
sage: OutputLatex in supp
True
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_ipython import␣
→BackendIPythonCommandline
>>> backend = BackendIPythonCommandline()
>>> supp = backend.supported_output();  supp      # random output
set([<class 'sage.repl.rich_output.output_graphics.OutputImageGif'>,
     ...,
     <class 'sage.repl.rich_output.output_graphics.OutputImagePng'>])
>>> from sage.repl.rich_output.output_basic import OutputLatex
>>> OutputLatex in supp
True
```

threejs_offline_scripts()

> Three.js script for the IPython command line.
>
> OUTPUT: string containing script tag
>
> EXAMPLES:

```
sage: # needs threejs
sage: from sage.repl.rich_output.backend_ipython import␣
```

```
→BackendIPythonCommandline
sage: backend = BackendIPythonCommandline()
sage: backend.threejs_offline_scripts()
'...<script ...</script>...'
```

```
>>> from sage.all import *
>>> # needs threejs
>>> from sage.repl.rich_output.backend_ipython import␣
→BackendIPythonCommandline
>>> backend = BackendIPythonCommandline()
>>> backend.threejs_offline_scripts()
'...<script ...</script>...'
```

class sage.repl.rich_output.backend_ipython.**BackendIPythonNotebook**

>   Bases: *BackendIPython*

>   Backend for the IPython Notebook.

>   EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import BackendIPythonNotebook
sage: BackendIPythonNotebook()
IPython notebook
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_ipython import BackendIPythonNotebook
>>> BackendIPythonNotebook()
IPython notebook
```

>   **displayhook**(*plain_text*, *rich_output*)

>   >   Backend implementation of the displayhook.

>   >   INPUT:

>   >   - plain_text – instance of *OutputPlainText*. The plain text version of the output.

>   >   - rich_output – instance of an output container class (subclass of *OutputBase*). Guaranteed to be one of the output containers returned from *supported_output()*, possibly the same as plain_text.

>   >   OUTPUT:

>   >   The IPython notebook display hook returns the IPython display data, a pair of dictionaries. The first dictionary contains mime types as keys and the respective output as value. The second dictionary is metadata.

>   >   EXAMPLES:

```
sage: from sage.repl.rich_output.output_basic import OutputPlainText
sage: plain_text = OutputPlainText.example()
sage: from sage.repl.rich_output.backend_ipython import BackendIPythonNotebook
sage: backend = BackendIPythonNotebook()
sage: backend.displayhook(plain_text, plain_text)
({'text/plain': 'Example plain text output'}, {})
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.output_basic import OutputPlainText
```

```
>>> plain_text = OutputPlainText.example()
>>> from sage.repl.rich_output.backend_ipython import BackendIPythonNotebook
>>> backend = BackendIPythonNotebook()
>>> backend.displayhook(plain_text, plain_text)
({'text/plain': 'Example plain text output'}, {})
```

supported_output()

Return the outputs that are supported by the IPython notebook backend.

OUTPUT:

Iterable of output container classes, that is, subclass of `OutputBase`). The order is ignored.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import BackendIPythonNotebook
sage: backend = BackendIPythonNotebook()
sage: supp = backend.supported_output();  supp       # random output
set([<class 'sage.repl.rich_output.output_graphics.OutputPlainText'>,
     ...,
     <class 'sage.repl.rich_output.output_graphics.OutputImagePdf'>])
sage: from sage.repl.rich_output.output_basic import OutputLatex
sage: OutputLatex in supp
True
sage: from sage.repl.rich_output.output_graphics import OutputImageGif
sage: OutputImageGif in supp
True
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_ipython import BackendIPythonNotebook
>>> backend = BackendIPythonNotebook()
>>> supp = backend.supported_output();  supp       # random output
set([<class 'sage.repl.rich_output.output_graphics.OutputPlainText'>,
     ...,
     <class 'sage.repl.rich_output.output_graphics.OutputImagePdf'>])
>>> from sage.repl.rich_output.output_basic import OutputLatex
>>> OutputLatex in supp
True
>>> from sage.repl.rich_output.output_graphics import OutputImageGif
>>> OutputImageGif in supp
True
```

threejs_offline_scripts()

Three.js script for the IPython notebook.

OUTPUT: string containing script tag

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import BackendIPythonNotebook
sage: backend = BackendIPythonNotebook()
sage: backend.threejs_offline_scripts()                                         #␣
↪needs sage.plot
'...<script src="/nbextensions/threejs-sage/r.../three.min.js...<\/script>...'
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output.backend_ipython import BackendIPythonNotebook
>>> backend = BackendIPythonNotebook()
>>> backend.threejs_offline_scripts()                                          #␣
↪needs sage.plot
'...<script src="/nbextensions/threejs-sage/r.../three.min.js...<\/script>...'
```

MISCELLANEOUS

6.1 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
- *SageTerminalInteractiveShell*
- *interface_shell_embed()*

6.1.1 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 *SageTerminalInteractiveShell*.
- 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.

6.1.2 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
```

```
>>> from sage.all import *
>>> shell = get_ipython()    # not tested
```

Any input is preprocessed and evaluated inside the `shell.run_cell` method. If the command line processing does not do what you want it to do, you can step through it in the debugger:

```
sage: %debug shell.run_cell('?')        # not tested
```

```
>>> from sage.all import *
>>> %debug shell.run_cell('?')        # not tested
```

The `SageInteractiveShell` provides the following customizations:

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

6.1.3 SageTerminalInteractiveShell

The *SageTerminalInteractiveShell* is a close relative of `SageInteractiveShell` that is specialized for running in a terminal. In particular, running commands like `!ls` will directly write to stdout. Technically, the `system` attribute will point to `system_raw` instead of `system_piped`.

6.1.4 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 *InterfaceShell-Transformer*.

class `sage.repl.interpreter.`**`InterfaceShellTransformer`**(***kwargs: Any*)

Bases: `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()*

`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 `max-ima(object)` if `shell.interface` is `maxima`.

- `line` – string; the line to transform

EXAMPLES:

```
sage: # needs sage.symbolic
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=shell.prefilter_manager)
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)') # maxima calls set_seed␣
↪on startup which is why 'sage0' will becomes 'sage4' and not just 'sage1'
'2 +  sage4 '
```

(continues on next page)

```
sage: ift.preparse_imports_from_sage('2 + gap(a)')
'2 + gap(a)'
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> from sage.repl.interpreter import interface_shell_embed,␣
↪InterfaceShellTransformer
>>> shell = interface_shell_embed(maxima)
>>> ift = InterfaceShellTransformer(shell=shell, config=shell.config,
...     prefilter_manager=shell.prefilter_manager)
>>> ift.shell.ex('a = 3')
>>> ift.preparse_imports_from_sage('2 + sage(a)')
'2 + sage0 '
>>> maxima.eval('sage0')
'3'
>>> ift.preparse_imports_from_sage('2 + maxima(a)') # maxima calls set_seed␣
↪on startup which is why 'sage0' will becomes 'sage4' and not just 'sage1'
'2 +  sage4 '
>>> ift.preparse_imports_from_sage('2 + gap(a)')
'2 + gap(a)'
```

Since Issue #28439, this also works with more complicated expressions containing nested parentheses:

```
sage: # needs sage.libs.gap sage.symbolic
sage: shell = interface_shell_embed(gap)
sage: shell.user_ns = locals()
sage: ift = InterfaceShellTransformer(shell=shell, config=shell.config,
....:     prefilter_manager=shell.prefilter_manager)
sage: line = '2 + sage((1+2)*gap(-(5-3)^2).sage()) - gap(1+(2-1))'
sage: line = ift.preparse_imports_from_sage(line)
sage: gap.eval(line)
'-12'
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap sage.symbolic
>>> shell = interface_shell_embed(gap)
>>> shell.user_ns = locals()
>>> ift = InterfaceShellTransformer(shell=shell, config=shell.config,
...     prefilter_manager=shell.prefilter_manager)
>>> line = '2 + sage((1+2)*gap(-(5-3)^2).sage()) - gap(1+(2-1))'
>>> line = ift.preparse_imports_from_sage(line)
>>> gap.eval(line)
'-12'
```

priority = 50

transform(*line*, *continue_prompt*)

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

- `line` – string; the line to be transformed *and evaluated*

- `continue_prompt` – boolean; whether this line is a continuation in a sequence of multiline input

EXAMPLES:

```
sage: # needs sage.symbolic
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=shell.prefilter_manager)
sage: ift.transform('2+2', False)   # note: output contains triple quotation␣
→marks
'sage.repl.interpreter.logstr(r"""4""")'
sage: ift.shell.ex('a = 4')
sage: ift.transform(r'sage(a)+4', False)
'sage.repl.interpreter.logstr(r"""8""")'
sage: ift.temporary_objects
set()
sage: shell = interface_shell_embed(gap)                                    #␣
→needs sage.libs.gap
sage: ift = InterfaceShellTransformer(shell=shell, config=shell.config,     #␣
→needs sage.libs.gap
....:     prefilter_manager=shell.prefilter_manager)
sage: ift.transform('2+2', False)
'sage.repl.interpreter.logstr(r"""4""")'
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> from sage.repl.interpreter import interface_shell_embed,␣
→InterfaceShellTransformer
>>> shell = interface_shell_embed(maxima)
>>> ift = InterfaceShellTransformer(shell=shell, config=shell.config,
...     prefilter_manager=shell.prefilter_manager)
>>> ift.transform('2+2', False)   # note: output contains triple quotation␣
→marks
'sage.repl.interpreter.logstr(r"""4""")'
>>> ift.shell.ex('a = 4')
>>> ift.transform(r'sage(a)+4', False)
'sage.repl.interpreter.logstr(r"""8""")'
>>> ift.temporary_objects
set()
>>> shell = interface_shell_embed(gap)                                      #␣
→needs sage.libs.gap
>>> ift = InterfaceShellTransformer(shell=shell, config=shell.config,       #␣
→needs sage.libs.gap
...     prefilter_manager=shell.prefilter_manager)
>>> ift.transform('2+2', False)
'sage.repl.interpreter.logstr(r"""4""")'
```

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

Bases: `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
```

(continues on next page)

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

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

class sage.repl.interpreter.**SageNotebookInteractiveShell**(***kwargs: Any*)

> Bases: *SageShellOverride*, InteractiveShell
>
> IPython Shell for the Sage IPython Notebook.
>
> The doctests are not tested since they would change the current rich output backend away from the doctest rich output backend.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.interpreter import SageNotebookInteractiveShell
> sage: SageNotebookInteractiveShell()    # not tested
> <sage.repl.interpreter.SageNotebookInteractiveShell object at 0x...>
> ```
>
> ```
> >>> from sage.all import *
> >>> from sage.repl.interpreter import SageNotebookInteractiveShell
> >>> SageNotebookInteractiveShell()    # not tested
> <sage.repl.interpreter.SageNotebookInteractiveShell object at 0x...>
> ```
>
> **init_display_formatter**()
>
> > Switch to the Sage IPython notebook rich output backend.
> >
> > EXAMPLES:
> >
> > ```
> > sage: from sage.repl.interpreter import SageNotebookInteractiveShell
> > sage: SageNotebookInteractiveShell().init_display_formatter()    # not tested
> > ```
> >
> > ```
> > >>> from sage.all import *
> > >>> from sage.repl.interpreter import SageNotebookInteractiveShell
> > >>> SageNotebookInteractiveShell().init_display_formatter()    # not tested
> > ```

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

>   EXAMPLES:

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

```
>>> from sage.all import *
>>> from sage.repl.interpreter import SagePreparseTransformer
>>> spt = SagePreparseTransformer
>>> spt(['1+1r+2.3^2.3r\n'])
["Integer(1)+1+RealNumber('2.3')**2.3\n"]
>>> preparser(False)
>>> spt(['2.3^2\n'])
['2.3^2\n']
```

> **ⓘ Note**
>
> IPython may call this function more than once for the same input lines. So when debugging the preparser, print outs may be duplicated. If using IPython >= 7.17, try: `sage.repl.interpreter.SagePreparseTransformer.has_side_effects = True`

class sage.repl.interpreter.**SageShellOverride**

>   Bases: `object`
>
>   Mixin to override methods in IPython's [Terminal]InteractiveShell classes.
>
>   **show_usage**()
>
> >   Print the basic Sage usage.
> >
> >   This method ends up being called when you enter ? and nothing else on the command line.
> >
> >   EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('?')
Welcome to Sage ...
sage: shell.quit()
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell()
>>> shell.run_cell('?')
Welcome to Sage ...
>>> shell.quit()
```

>   **system_raw**(*cmd*)
>
> >   Run a system command.

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('R --version')    # optional - r
R version ...
sage: shell.user_ns['_exit_code']        # optional - r
0
sage: shell.quit()
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell()
>>> shell.system_raw('false')
>>> shell.user_ns['_exit_code'] > Integer(0)
True
>>> shell.system_raw('true')
>>> shell.user_ns['_exit_code']
0
>>> shell.system_raw('R --version')    # optional - r
R version ...
>>> shell.user_ns['_exit_code']        # optional - r
0
>>> shell.quit()
```

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

    Bases: TerminalIPythonApp

    crash_handler_class

        alias of *SageCrashHandler*

    init_shell()

        Initialize the SageInteractiveShell instance.

> **ⓘ Note**
>
> This code is based on TerminalIPythonApp.init_shell().

        EXAMPLES:

```
sage: from sage.repl.interpreter import SageTerminalApp
sage: app = SageTerminalApp.instance()
sage: app.shell
<sage.repl.interpreter.SageTestShell object at 0x...>
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import SageTerminalApp
```

```
>>> app = SageTerminalApp.instance()
>>> app.shell
<sage.repl.interpreter.SageTestShell 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()`.

name: str | Unicode[str, str | bytes] = 'Sage'

shell_class

Type of the shell

test_shell

Whether the shell is a test shell

class sage.repl.interpreter.**SageTerminalInteractiveShell**(***kwargs: Any*)

Bases: *SageShellOverride*, TerminalInteractiveShell

IPython Shell for the Sage IPython Commandline Interface.

The doctests are not tested since they would change the current rich output backend away from the doctest rich output backend.

EXAMPLES:

```
sage: from sage.repl.interpreter import SageTerminalInteractiveShell
sage: SageTerminalInteractiveShell()   # not tested
<sage.repl.interpreter.SageNotebookInteractiveShell object at 0x...>
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import SageTerminalInteractiveShell
>>> SageTerminalInteractiveShell()   # not tested
<sage.repl.interpreter.SageNotebookInteractiveShell object at 0x...>
```

init_display_formatter()

Switch to the Sage IPython commandline rich output backend.

EXAMPLES:

```
sage: from sage.repl.interpreter import SageTerminalInteractiveShell
sage: SageTerminalInteractiveShell().init_display_formatter()   # not tested
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import SageTerminalInteractiveShell
>>> SageTerminalInteractiveShell().init_display_formatter()   # not tested
```

prompt_for_code()

class sage.repl.interpreter.**SageTestShell**(***kwargs: Any*)

>   Bases: *SageShellOverride*, TerminalInteractiveShell
>
>   Test Shell.
>
>   Care must be taken in these doctests to quit the test shell in order to switch back the rich output display backend to the doctest backend.
>
>   EXAMPLES:
>
> ```
> sage: from sage.repl.interpreter import get_test_shell
> sage: shell = get_test_shell();   shell
> <sage.repl.interpreter.SageTestShell object at 0x...>
> sage: shell.quit()
> ```
>
> ```
> >>> from sage.all import *
> >>> from sage.repl.interpreter import get_test_shell
> >>> shell = get_test_shell();   shell
> <sage.repl.interpreter.SageTestShell object at 0x...>
> >>> shell.quit()
> ```
>
>   **init_display_formatter**()
>
>   >   Switch to the Sage IPython commandline rich output backend.
>   >
>   >   EXAMPLES:
>   >
>    > ```
>    > sage: from sage.repl.interpreter import get_test_shell
>    > sage: shell = get_test_shell();   shell
>    > <sage.repl.interpreter.SageTestShell object at 0x...>
>    > sage: shell.quit()
>    > sage: shell.init_display_formatter()
>    > sage: shell.quit()
>    > ```
>   >
>    > ```
>    > >>> from sage.all import *
>    > >>> from sage.repl.interpreter import get_test_shell
>    > >>> shell = get_test_shell();   shell
>    > <sage.repl.interpreter.SageTestShell object at 0x...>
>    > >>> shell.quit()
>    > >>> shell.init_display_formatter()
>    > >>> shell.quit()
>    > ```
>
>   **quit**()
>
>   >   Quit the test shell.
>   >
>   >   To make the test shell as realistic as possible, we switch to the *BackendIPythonCommandline* display backend. This method restores the previous display backend, which is the *BackendDoctest* during doctests.
>   >
>   >   EXAMPLES:
>   >
>    > ```
>    > sage: from sage.repl.interpreter import get_test_shell
>    > sage: from sage.repl.rich_output import get_display_manager
>    > sage: get_display_manager()
>    > The Sage display manager using the doctest backend
>   >
>   > sage: shell = get_test_shell()
>    > ```
>   >
>   >

```
sage: get_display_manager()
The Sage display manager using the IPython command line backend

sage: shell.quit()
sage: get_display_manager()
The Sage display manager using the doctest backend
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell
>>> from sage.repl.rich_output import get_display_manager
>>> get_display_manager()
The Sage display manager using the doctest backend

>>> shell = get_test_shell()
>>> get_display_manager()
The Sage display manager using the IPython command line backend

>>> shell.quit()
>>> get_display_manager()
The Sage display manager using the doctest backend
```

run_cell(*args*, ***kwds*)

Run IPython cell.

Starting with IPython-3.0, this returns an success/failure information. Since it is more convenient for doctests, we ignore it.

EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: rc = shell.run_cell('2^50')
1125899906842624
sage: rc is None
True
sage: shell.quit()
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell()
>>> rc = shell.run_cell('2^50')
1125899906842624
>>> rc is None
True
>>> shell.quit()
```

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

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

OUTPUT: an IPython shell

EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell(); shell
<sage.repl.interpreter.SageTestShell object at 0x...>
sage: shell.parent.shell_class
<class 'sage.repl.interpreter.SageTestShell'>
sage: shell.parent.test_shell
True
sage: shell.quit()
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell(); shell
<sage.repl.interpreter.SageTestShell object at 0x...>
>>> shell.parent.shell_class
<class 'sage.repl.interpreter.SageTestShell'>
>>> shell.parent.test_shell
True
>>> shell.quit()
```

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

> Return 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)                              #␣
↪needs sage.libs.gap
sage: shell.run_cell('List( [1..10], IsPrime )')                     #␣
↪needs sage.libs.gap
[ false, true, true, false, true, false, true, false, false, false ]
<ExecutionResult object at ..., execution_count=None error_before_exec=None error_
↪in_exec=None ...result=[ false, true, true, false, true, false, true, false,␣
↪false, false ]>
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import interface_shell_embed
>>> shell = interface_shell_embed(gap)                               #␣
↪needs sage.libs.gap
>>> shell.run_cell('List( [1..10], IsPrime )')                      #␣
↪needs sage.libs.gap
[ false, true, true, false, true, false, true, false, false, false ]
<ExecutionResult object at ..., execution_count=None error_before_exec=None error_
↪in_exec=None ...result=[ false, true, true, false, true, false, true, false,␣
↪false, false ]>
```

class sage.repl.interpreter.**logstr**

> Bases: `str`
>
> For use by :meth`~InterfaceShellTransformer.transform`. This provides a `_latex_` method which is just the string wrapped in a `\verb` environment.

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

    Turn on or off the Sage preparser.

        • `on` – boolean; whether to turn on preparsing

    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
```

```
>>> from sage.all import *
>>> Integer(2)/Integer(3)
2/3
>>> preparser(False)
>>> Integer(2)/Integer(3)  # not tested since doctests are always preparsed
0
>>> preparser(True)
>>> Integer(2)**Integer(3)
8
```

6.2 Sage's IPython Extension

A Sage extension which adds sage-specific features:

• magics

    – `%crun`

    – `%runfile`

    – `%attach`

    – `%display`

    – `%mode` (like `%maxima`, etc.)

    – `%%cython`

    – `%%fortran`

• preparsing of input

• loading Sage library

• running init.sage

• changing prompt to Sage prompt

• Display hook

class `sage.repl.ipython_extension.`**`SageCustomizations`**(*shell=None*)

    Bases: `object`

    Initialize the Sage plugin.

static all_globals()

Return a Python module containing all globals which should be made available to the user.

EXAMPLES:

```
sage: from sage.repl.ipython_extension import SageCustomizations
sage: SageCustomizations.all_globals()
<module 'sage.all_cmdline' ...>
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_extension import SageCustomizations
>>> SageCustomizations.all_globals()
<module 'sage.all_cmdline' ...>
```

init_environment()

Set up Sage command-line environment

init_inspector()

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.

class sage.repl.ipython_extension.**SageJupyterCustomizations**(*shell=None*)

Bases: *SageCustomizations*

static all_globals()

Return a Python module containing all globals which should be made available to the user when running the Jupyter notebook.

EXAMPLES:

```
sage: from sage.repl.ipython_extension import SageJupyterCustomizations
sage: SageJupyterCustomizations.all_globals()
<module 'sage.repl.ipython_kernel.all_jupyter' ...>
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_extension import SageJupyterCustomizations
>>> SageJupyterCustomizations.all_globals()
<module 'sage.repl.ipython_kernel.all_jupyter' ...>
```

class sage.repl.ipython_extension.**SageMagics**(***kwargs: Any*)

Bases: 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

> ↪ **See also**
>
> This is the same as *attach()*.

EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: from tempfile import NamedTemporaryFile as NTF
sage: with NTF(mode='w+t', suffix='.py', delete=False) as f:
....:     _ = f.write('a = 2\n')
sage: shell.run_cell('%attach ' + f.name)
sage: shell.run_cell('a')
2
sage: sleep(1)   # filesystem timestamp granularity
sage: with open(f.name, 'w') as f: _ = f.write('a = 3\n')
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell()
>>> from tempfile import NamedTemporaryFile as NTF
>>> with NTF(mode='w+t', suffix='.py', delete=False) as f:
...     _ = f.write('a = 2\n')
>>> shell.run_cell('%attach ' + f.name)
>>> shell.run_cell('a')
2
>>> sleep(Integer(1))   # filesystem timestamp granularity
>>> with open(f.name, 'w') as f: _ = f.write('a = 3\n')
```

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.attach import reload_attached_files_if_
↪modified')
sage: shell.run_cell('reload_attached_files_if_modified()')
### reloading attached file ... modified at ... ###

sage: shell.run_cell('a')
3
sage: shell.run_cell('detach(%r)' % f.name)
sage: shell.run_cell('attached_files()')
[]
sage: os.remove(f.name)
sage: shell.quit()
```

```
>>> from sage.all import *
>>> shell.run_cell('from sage.repl.attach import reload_attached_files_if_
↪modified')
>>> shell.run_cell('reload_attached_files_if_modified()')
### reloading attached file ... modified at ... ###

>>> shell.run_cell('a')
3
```

```
>>> shell.run_cell('detach(%r)' % f.name)
>>> shell.run_cell('attached_files()')
[]
>>> os.remove(f.name)
>>> shell.quit()
```

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 ...
sage: shell.quit()
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell()
>>> 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 ...
>>> shell.quit()
```

cython(*line*, *cell*)

Cython cell magic.

This is syntactic sugar on the `cython_compile()` function.

Note that there is also the `%%cython` cell magic provided by Cython, which can be loaded with `%load_ext cython`, see Cython documentation for more details. The semantic is slightly different from the version provided by Sage.

INPUT:

- `line` – parsed as keyword arguments. The allowed arguments are:

    - `--verbose N` / `-v N`

    - `--compile-message`

    - `--use-cache`

    - `--create-local-c-file`

    - `--annotate`

    - `--view-annotate`

> > – `--sage-namespace`
>
> > – `--create-local-so-file`
>
> > – `--no-compile-message`, `--no-use-cache`, etc.
>
> See `cython()` for details.
>
> If `--view-annotate` is given, the annotation is either displayed inline in the Sage notebook or opened in a new web browser, depending on whether the Sage notebook is used.
>
> You can override the selection by specifying `--view-annotate=webbrowser` or `--view-anno-tate=displayhtml`.
>
> • `cell` – string; the Cython source code to process

OUTPUT: none; the Cython code is compiled and loaded

EXAMPLES:

```
sage: # needs sage.misc.cython
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell(
....: '''
....: %%cython -v1 --annotate --no-sage-namespace
....: def f():
....:     print('test')
....: ''')
Compiling ....pyx because it changed.
[1/1] Cythonizing ....pyx
sage: f()
test
```

```
>>> from sage.all import *
>>> # needs sage.misc.cython
>>> from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell()
>>> shell.run_cell(
... '''
... %%cython -v1 --annotate --no-sage-namespace
... def f():
...     print('test')
... ''')
Compiling ....pyx because it changed.
[1/1] Cythonizing ....pyx
>>> f()
test
```

display(*args*)

> A magic command to switch between simple display and ASCII art display.
>
> • `args` – string. See *sage.repl.rich_output.preferences* for allowed values. If the mode is `ascii_art`, it can optionally be followed by a width.
>
> How to use: if you want to activate the ASCII art mode:

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

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

That means you do not have to use `ascii_art()` to get an ASCII art output:

```
sage: shell.run_cell("i = var('i')")                                    #␣
↪needs sage.symbolic
sage: shell.run_cell('sum(i^2*x^i, i, 0, 10)')                          #␣
↪needs sage.symbolic
    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
```

```
>>> from sage.all import *
>>> shell.run_cell("i = var('i')")                                      #␣
↪needs sage.symbolic
>>> shell.run_cell('sum(i^2*x^i, i, 0, 10)')                            #␣
↪needs sage.symbolic
    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 to return to 'textual mode':

```
sage: shell.run_cell('%display text plain')
sage: shell.run_cell('%display plain')          # shortcut for "text plain"
sage: shell.run_cell('sum(i^2*x^i, i, 0, 10)')                          #␣
↪needs sage.symbolic
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
```

```
>>> from sage.all import *
>>> shell.run_cell('%display text plain')
>>> shell.run_cell('%display plain')          # shortcut for "text plain"
>>> shell.run_cell('sum(i^2*x^i, i, 0, 10)')                            #␣
↪needs sage.symbolic
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()')                      #␣
↪needs sage.combinat
[
[                                                               1  4   1 ␣
↪3
[                 1  3  4    1  2  4    1  2  3    1  3    1  2   2      2
```

(continues on next page)

```
[   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()')                              #␣
↪needs sage.combinat
[
[
[                   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 ]
```

```
>>> from sage.all import *
>>> shell.run_cell('%display ascii_art')
>>> shell.run_cell('StandardTableaux(4).list()')                                #␣
↪needs sage.combinat
[
[                                                          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 ␣
↪ ,
<BLANKLINE>
              1 ]
   1  2     2 ]
   3        3 ]
   4    ,   4 ]
>>> shell.run_cell('%display ascii_art 50')
>>> shell.run_cell('StandardTableaux(4).list()')                                #␣
↪needs sage.combinat
[
[
[                   1  3  4     1  2  4     1  2  3
[   1  2  3  4,     2       ,   3       ,   4       ,
<BLANKLINE>
                                              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 text latex')
sage: shell.run_cell('1/2')
```

```
1/2
```

```
>>> from sage.all import *
>>> shell.run_cell('%display text latex')
>>> shell.run_cell('1/2')
1/2
```

Switch back:

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

```
>>> from sage.all import *
>>> shell.run_cell('%display default')
```

Switch graphics to default to vector or raster graphics file formats:

```
sage: shell.run_cell('%display graphics vector')
```

```
>>> from sage.all import *
>>> shell.run_cell('%display graphics vector')
```

fortran(*line*, *cell*)

Fortran cell magic.

This is syntactic sugar on the `fortran()` function.

INPUT:

- `line` – ignored

- `cell` – string; the Cython source code to process

OUTPUT: none; the Fortran code is compiled and loaded

EXAMPLES:

```
sage: # needs numpy
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('''
....: %%fortran
....: C FILE: FIB1.F
....:       SUBROUTINE FIB(A,N)
....: C
....: C     CALCULATE FIRST N FIBONACCI NUMBERS
....: C
....:       INTEGER N
....:       REAL*8 A(N)
....:       DO I=1,N
....:           IF (I.EQ.1) THEN
....:               A(I) = 0.0D0
....:           ELSEIF (I.EQ.2) THEN
....:               A(I) = 1.0D0
....:           ELSE
```

```
....:                A(I) = A(I-1) + A(I-2)
....:            ENDIF
....:        ENDDO
....:        END
....: C END FILE FIB1.F
....: ''')
sage: fib
<fortran ...>
sage: from numpy import array
sage: a = array(range(10), dtype=float)
sage: fib(a, 10)
sage: a
array([  0.,   1.,   1.,   2.,   3.,   5.,   8.,  13.,  21.,  34.])
```

```
>>> from sage.all import *
>>> # needs numpy
>>> from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell()
>>> shell.run_cell('''
... %%fortran
... C FILE: FIB1.F
...       SUBROUTINE FIB(A,N)
... C
... C     CALCULATE FIRST N FIBONACCI NUMBERS
... C
...       INTEGER N
...       REAL*8 A(N)
...       DO I=1,N
...          IF (I.EQ.1) THEN
...             A(I) = 0.0D0
...          ELSEIF (I.EQ.2) THEN
...             A(I) = 1.0D0
...          ELSE
...             A(I) = A(I-1) + A(I-2)
...          ENDIF
...       ENDDO
...       END
... C END FILE FIB1.F
... ''')
>>> fib
<fortran ...>
>>> from numpy import array
>>> a = array(range(Integer(10)), dtype=float)
>>> fib(a, Integer(10))
>>> a
array([  0.,   1.,   1.,   2.,   3.,   5.,   8.,  13.,  21.,  34.])
```

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
```

```
>>> from sage.all import *
>>> ip = get_ipython()             # not tested: works only in interactive shell
>>> ip.magic_iload('/dev/null')   # not tested: works only in interactive shell
Interactively loading "/dev/null"  # not tested: works only in interactive␣
↪shell
```

**magics = {'cell': {'cython': 'cython', 'fortran': 'fortran'}, 'line': {'attach':
'attach', 'crun': 'crun', 'display': 'display', 'iload': 'iload', 'runfile':
'runfile'}}**

registered = True

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

> **↪ See also**
>
> This is the same as *load()*.

EXAMPLES:

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

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

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

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

   Load the extension in IPython.

6.3 Magics for each of the Sage interfaces

This module defines magic functions for interpreters. As an example, consider the GAP interpreter which can evaluate a gap command given as a string:

```
sage: gap('SymmetricGroup(4)')         # not tested
SymmetricGroup( [ 1 .. 4 ] )
```

```
>>> from sage.all import *
>>> gap('SymmetricGroup(4)')           # not tested
SymmetricGroup( [ 1 .. 4 ] )
```

Magics are syntactic sugar to avoid writing the Python string. They are either called as line magics:

```
sage: %gap SymmetricGroup(4)           # not tested
```

```
>>> from sage.all import *
>>> %gap SymmetricGroup(Integer(4))        # not tested
```

or as cell magics, that is, spanning multiple lines:

```
sage: %%gap                            # not tested
....: G := SymmetricGroup(4);
....: Display(G);
```

```
>>> from sage.all import *
>>> %%gap                              # not tested
... G := SymmetricGroup(Integer(4));
... Display(G);
```

Note that the cell magic needs semicolons, this is required by the GAP language to separate multiple commands.

class sage.repl.interface_magic.**InterfaceMagic**(*name*, *interface*)

   Bases: object

   Interface Magic.

   This class is a wrapper around interface objects to provide them with magics.

   INPUT:

- `name` – string; the interface name

- `interface` – `sage.interfaces.expect.Expect`; the interface to wrap

EXAMPLES:

```
sage: from sage.repl.interface_magic import InterfaceMagic
sage: InterfaceMagic.find('gap')                                    #
→needs sage.libs.gap
<sage.repl.interface_magic.InterfaceMagic object at 0x...>
```

```
>>> from sage.all import *
>>> from sage.repl.interface_magic import InterfaceMagic
>>> InterfaceMagic.find('gap')                                      # needs
→sage.libs.gap
<sage.repl.interface_magic.InterfaceMagic object at 0x...>
```

classmethod all_iter()

    Iterate over the available interfaces.

    EXAMPLES:

```
sage: from sage.repl.interface_magic import InterfaceMagic
sage: next(InterfaceMagic.all_iter())
<sage.repl.interface_magic.InterfaceMagic object at 0x...>
```

```
>>> from sage.all import *
>>> from sage.repl.interface_magic import InterfaceMagic
>>> next(InterfaceMagic.all_iter())
<sage.repl.interface_magic.InterfaceMagic object at 0x...>
```

cell_magic_factory()

    Factory for cell magic.

    OUTPUT: a function suitable to be used as cell magic

    EXAMPLES:

```
sage: # needs sage.libs.gap
sage: from sage.repl.interface_magic import InterfaceMagic
sage: cell_magic = InterfaceMagic.find('gap').cell_magic_factory()
sage: output = cell_magic('', '1+1;')
2
sage: output is None
True
sage: cell_magic('foo', '1+1;')
Traceback (most recent call last):
...
SyntaxError: Interface magics have no options, got "foo"
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> from sage.repl.interface_magic import InterfaceMagic
>>> cell_magic = InterfaceMagic.find('gap').cell_magic_factory()
>>> output = cell_magic('', '1+1;')
```

```
2
>>> output is None
True
>>> cell_magic('foo', '1+1;')
Traceback (most recent call last):
...
SyntaxError: Interface magics have no options, got "foo"
```

This is how the built cell magic is used in practice:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('%%gap\nG:=SymmetricGroup(5);\n1+1;Order(G);')          #␣
→needs sage.libs.gap
Sym( [ 1 .. 5 ] )
2
120
sage: shell.run_cell('%%gap foo\n1+1;\n')                                    #␣
→needs sage.libs.gap
...File...<string>...
SyntaxError: Interface magics have no options, got "foo"

sage: shell.run_cell('%%gap?')                                              #␣
→needs sage.libs.gap
Docstring:
Interact with gap

The cell magic %%gap sends multiple lines to the gap interface.
...
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell()
>>> shell.run_cell('%%gap\nG:=SymmetricGroup(5);\n1+1;Order(G);')          #␣
→needs sage.libs.gap
Sym( [ 1 .. 5 ] )
2
120
>>> shell.run_cell('%%gap foo\n1+1;\n')                                    #␣
→needs sage.libs.gap
...File...<string>...
SyntaxError: Interface magics have no options, got "foo"
<BLANKLINE>
>>> shell.run_cell('%%gap?')                                              #␣
→needs sage.libs.gap
Docstring:
Interact with gap
<BLANKLINE>
The cell magic %%gap sends multiple lines to the gap interface.
...
```

classmethod find(*name*)

> Find a particular magic by name.

This method is for doctesting purposes only.

INPUT:

- `name` – string; the name of the interface magic to search for

OUTPUT: the corresponding *InterfaceMagic* instance

EXAMPLES:

```
sage: from sage.repl.interface_magic import InterfaceMagic
sage: InterfaceMagic.find('gap')                                          #␣
↪needs sage.libs.gap
<sage.repl.interface_magic.InterfaceMagic object at 0x...>
```

```
>>> from sage.all import *
>>> from sage.repl.interface_magic import InterfaceMagic
>>> InterfaceMagic.find('gap')                                            #␣
↪needs sage.libs.gap
<sage.repl.interface_magic.InterfaceMagic object at 0x...>
```

line_magic_factory()

Factory for line magic.

OUTPUT: a function suitable to be used as line magic

EXAMPLES:

```
sage: # needs sage.libs.gap
sage: from sage.repl.interface_magic import InterfaceMagic
sage: line_magic = InterfaceMagic.find('gap').line_magic_factory()
sage: output = line_magic('1+1')
sage: output
2
sage: type(output)
<class 'sage.interfaces.gap.GapElement'>
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> from sage.repl.interface_magic import InterfaceMagic
>>> line_magic = InterfaceMagic.find('gap').line_magic_factory()
>>> output = line_magic('1+1')
>>> output
2
>>> type(output)
<class 'sage.interfaces.gap.GapElement'>
```

This is how the built line magic is used in practice:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('%gap 1+1')                                          #␣
↪needs sage.libs.gap
2
sage: shell.run_cell('%gap?')                                            #␣
↪needs sage.libs.gap
```

```
Docstring:
Interact with gap

The line magic %gap sends a single line to the gap interface.
...
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell()
>>> shell.run_cell('%gap 1+1')                                      #␣
↪needs sage.libs.gap
2
>>> shell.run_cell('%gap?')                                         #␣
↪needs sage.libs.gap
Docstring:
Interact with gap
<BLANKLINE>
The line magic %gap sends a single line to the gap interface.
...
```

classmethod register_all(*shell=None*)

Register all available interfaces.

EXAMPLES:

```
sage: class MockShell():
....:     magics = set()
....:     def register_magic_function(self, fn, magic_name, magic_kind):
....:         self.magics.add(magic_name)
....:         print(magic_name, magic_kind)
sage: from sage.repl.interface_magic import InterfaceMagic
sage: InterfaceMagic.register_all(MockShell())    # random output
('gp', 'line')
('gp', 'cell')
('mwrank', 'line')
('mwrank', 'cell')
...
('maxima', 'line')
('maxima', 'cell')
sage: 'gap' in MockShell.magics                                    #␣
↪needs sage.libs.gap
True
sage: 'maxima' in MockShell.magics                                 #␣
↪needs sage.symbolic
True
```

```
>>> from sage.all import *
>>> class MockShell():
...     magics = set()
...     def register_magic_function(self, fn, magic_name, magic_kind):
...         self.magics.add(magic_name)
...         print(magic_name, magic_kind)
```

```
>>> from sage.repl.interface_magic import InterfaceMagic
>>> InterfaceMagic.register_all(MockShell())     # random output
('gp', 'line')
('gp', 'cell')
('mwrank', 'line')
('mwrank', 'cell')
...
('maxima', 'line')
('maxima', 'cell')
>>> 'gap' in MockShell.magics                                      #␣
→needs sage.libs.gap
True
>>> 'maxima' in MockShell.magics                                   #␣
→needs sage.symbolic
True
```

6.4 Interacts for the Sage Jupyter notebook

This is mostly the same as the stock `ipywidgets.interact`, but with some customizations for Sage.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.interact import interact
sage: @interact
....: def f(x=(0, 10)):
....:     pass
...Interactive function <function f at ...> with 1 widget
  x: IntSlider(value=5, description='x', max=10)
sage: f.widget.children
(IntSlider(value=5, description='x', max=10), Output())
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.interact import interact
>>> @interact
... def f(x=(Integer(0), Integer(10))):
...     pass
...Interactive function <function f at ...> with 1 widget
  x: IntSlider(value=5, description='x', max=10)
>>> f.widget.children
(IntSlider(value=5, description='x', max=10), Output())
```

class sage.repl.ipython_kernel.interact.**sage_interactive**(***kwargs: Any*)

> Bases: `interactive`
>
> Wrapper around the ipywidgets interactive which handles some SageNB specifics.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.ipython_kernel.interact import sage_interactive
> sage: def myfunc(x=10, y='hello', z=None): pass
> sage: sage_interactive(myfunc, x=(0,100), z=["one", "two", "three"])
> ...Interactive function <function myfunc at ...> with 3 widgets
> ```

```
  x: IntSlider(value=10, description='x')
  y: Text(value='hello', description='y')
  z: Dropdown(description='z', options=('one', 'two', 'three'), value=None)
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.interact import sage_interactive
>>> def myfunc(x=Integer(10), y='hello', z=None): pass
>>> sage_interactive(myfunc, x=(Integer(0),Integer(100)), z=["one", "two", "three
↪"])
...Interactive function <function myfunc at ...> with 3 widgets
  x: IntSlider(value=10, description='x')
  y: Text(value='hello', description='y')
  z: Dropdown(description='z', options=('one', 'two', 'three'), value=None)
```

signature()

> Return the fixed signature of the interactive function (after a possible `auto_update` parameter was removed).
>
> EXAMPLES:

```
sage: from sage.repl.ipython_kernel.interact import sage_interactive
sage: def myfunc(x=[1,2,3], auto_update=False): pass
sage: sage_interactive(myfunc).signature().parameters
...mappingproxy({'x': <Parameter "x=[1, 2, 3]">})
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.interact import sage_interactive
>>> def myfunc(x=[Integer(1),Integer(2),Integer(3)], auto_update=False): pass
>>> sage_interactive(myfunc).signature().parameters
...mappingproxy({'x': <Parameter "x=[1, 2, 3]">})
```

classmethod widget_from_iterable(*abbrev*, **args*, ***kwds*)

> Convert an unspecified iterable to a widget.
>
> This behaves like in ipywidgets, except that an iterator (like a generator object) becomes a `Selection-Slider`.
>
> EXAMPLES:

```
sage: from sage.repl.ipython_kernel.interact import sage_interactive
sage: sage_interactive.widget_from_iterable([1..5])
...Dropdown(options=(1, 2, 3, 4, 5), value=1)
sage: sage_interactive.widget_from_iterable(iter([1..5]))
...SelectionSlider(options=(1, 2, 3, 4, 5), value=1)
sage: sage_interactive.widget_from_iterable((1..5))
...SelectionSlider(options=(1, 2, 3, 4, 5), value=1)
sage: sage_interactive.widget_from_iterable(x for x in [1..5])
...SelectionSlider(options=(1, 2, 3, 4, 5), value=1)
sage: def gen():
....:     yield 1; yield 2; yield 3; yield 4; yield 5
sage: sage_interactive.widget_from_iterable(gen())
...SelectionSlider(options=(1, 2, 3, 4, 5), value=1)
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.interact import sage_interactive
>>> sage_interactive.widget_from_iterable((ellipsis_range(Integer(1),Ellipsis,
→Integer(5))))
...Dropdown(options=(1, 2, 3, 4, 5), value=1)
>>> sage_interactive.widget_from_iterable(iter((ellipsis_range(Integer(1),
→Ellipsis,Integer(5)))))
...SelectionSlider(options=(1, 2, 3, 4, 5), value=1)
>>> sage_interactive.widget_from_iterable((ellipsis_iter(Integer(1),Ellipsis,
→Integer(5))))
...SelectionSlider(options=(1, 2, 3, 4, 5), value=1)
>>> sage_interactive.widget_from_iterable(x for x in (ellipsis_
→range(Integer(1),Ellipsis,Integer(5))))
...SelectionSlider(options=(1, 2, 3, 4, 5), value=1)
>>> def gen():
...     yield Integer(1); yield Integer(2); yield Integer(3); yield_
→Integer(4); yield Integer(5)
>>> sage_interactive.widget_from_iterable(gen())
...SelectionSlider(options=(1, 2, 3, 4, 5), value=1)
```

classmethod **widget_from_single_value**(*abbrev*, **args*, ***kwds*)

Convert a single value (i.e. a non-iterable) to a widget.

This supports the Sage `Color` and `Matrix` classes. Any unknown type is changed to a string for evaluating. This is meant to support symbolic expressions like `sin(x)`.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.interact import sage_interactive
sage: sage_interactive.widget_from_single_value("sin(x)")
...Text(value='sin(x)')
sage: sage_interactive.widget_from_single_value(sin(x))                        #_
→needs sage.symbolic
...EvalText(value='sin(x)')
sage: sage_interactive.widget_from_single_value(matrix([[1, 2], [3, 4]]))    #_
→needs sage.modules
...Grid(value=[[1, 2], [3, 4]], children=(Label(value=''),_
→VBox(children=(EvalText(value='1', layout=Layout(max_width='5em')),_
→EvalText(value='3', layout=Layout(max_width='5em')))),_
→VBox(children=(EvalText(value='2', layout=Layout(max_width='5em')),_
→EvalText(value='4', layout=Layout(max_width='5em')))))))
sage: from sage.plot.colors import Color                                       #_
→needs sage.plot
sage: sage_interactive.widget_from_single_value(Color('cornflowerblue'))     #_
→needs sage.plot
...SageColorPicker(value='#6495ed')
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.interact import sage_interactive
>>> sage_interactive.widget_from_single_value("sin(x)")
...Text(value='sin(x)')
>>> sage_interactive.widget_from_single_value(sin(x))                          #_
→needs sage.symbolic
```

<div align="right">(continues on next page)</div>

```
...EvalText(value='sin(x)')
>>> sage_interactive.widget_from_single_value(matrix([[Integer(1),␣
↪Integer(2)], [Integer(3), Integer(4)]]))   # needs sage.modules
...Grid(value=[[1, 2], [3, 4]], children=(Label(value=''),␣
↪VBox(children=(EvalText(value='1', layout=Layout(max_width='5em')),␣
↪EvalText(value='3', layout=Layout(max_width='5em')))),␣
↪VBox(children=(EvalText(value='2', layout=Layout(max_width='5em')),␣
↪EvalText(value='4', layout=Layout(max_width='5em')))))))
>>> from sage.plot.colors import Color                                   #␣
↪needs sage.plot
>>> sage_interactive.widget_from_single_value(Color('cornflowerblue'))    #␣
↪needs sage.plot
...SageColorPicker(value='#6495ed')
```

classmethod widget_from_tuple(*abbrev*, **args*, ***kwds*)

Convert a tuple to a widget.

This supports two SageNB extensions: `(description,  abbrev)` if `description` is a string and `(default, abbrev)` if `abbrev` is not a single value.

Symbolic expressions are changed to a floating-point number.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.interact import sage_interactive
sage: sage_interactive.widget_from_tuple( (0, 10) )
...IntSlider(value=5, max=10)
sage: sage_interactive.widget_from_tuple( ("number", (0, 10)) )
...IntSlider(value=5, description='number', max=10)
sage: sage_interactive.widget_from_tuple( (3, (0, 10)) )
...IntSlider(value=3, max=10)
sage: sage_interactive.widget_from_tuple((2, [('one', 1), ('two', 2), ('three
↪', 3)]))
...Dropdown(index=1, options=(('one', 1), ('two', 2), ('three', 3)), value=2)
sage: sage_interactive.widget_from_tuple( (sqrt(2), pi) )                 #␣
↪needs sage.symbolic
...FloatSlider(value=2.277903107981444, max=3.141592653589793, min=1.
↪4142135623730951)
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.interact import sage_interactive
>>> sage_interactive.widget_from_tuple( (Integer(0), Integer(10)) )
...IntSlider(value=5, max=10)
>>> sage_interactive.widget_from_tuple( ("number", (Integer(0), Integer(10)))␣
↪)
...IntSlider(value=5, description='number', max=10)
>>> sage_interactive.widget_from_tuple( (Integer(3), (Integer(0),␣
↪Integer(10))) )
...IntSlider(value=3, max=10)
>>> sage_interactive.widget_from_tuple((Integer(2), [('one', Integer(1)), (
↪'two', Integer(2)), ('three', Integer(3))]))
...Dropdown(index=1, options=(('one', 1), ('two', 2), ('three', 3)), value=2)
>>> sage_interactive.widget_from_tuple( (sqrt(Integer(2)), pi) )          ␣
```

```
→      # needs sage.symbolic
...FloatSlider(value=2.277903107981444, max=3.141592653589793, min=1.
→4142135623730951)
```

6.5 Widgets to be used for the Sage Jupyter notebook

These are all based on widgets from `ipywidgets`, changing or combining existing widgets.

class sage.repl.ipython_kernel.widgets.**EvalText**(***kwargs: Any*)

> Bases: *EvalWidget*, Text
>
> A `ipywidgets.Text` widget which evaluates (using `sage_eval()`) its contents and applies an optional transformation.
>
> EXAMPLES:

```
sage: from sage.repl.ipython_kernel.widgets import EvalText
sage: w = EvalText(value='pi', transform=lambda x: x^2)
sage: w
EvalText(value='pi')
sage: w.get_interact_value()                                             #␣
→needs sage.symbolic
pi^2
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.widgets import EvalText
>>> w = EvalText(value='pi', transform=lambda x: x**Integer(2))
>>> w
EvalText(value='pi')
>>> w.get_interact_value()                                               #␣
→needs sage.symbolic
pi^2
```

class sage.repl.ipython_kernel.widgets.**EvalTextarea**(***kwargs: Any*)

> Bases: *EvalWidget*, Textarea
>
> A `ipywidgets.Textarea` widget which evaluates (using `sage_eval()`) its contents and applies an optional transformation.
>
> EXAMPLES:

```
sage: from sage.repl.ipython_kernel.widgets import EvalTextarea
sage: w = EvalTextarea(value='pi', transform=lambda x: x^2)
sage: w
EvalTextarea(value='pi')
sage: w.get_interact_value()                                             #␣
→needs sage.symbolic
pi^2
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.widgets import EvalTextarea
>>> w = EvalTextarea(value='pi', transform=lambda x: x**Integer(2))
>>> w
```

```
EvalTextarea(value='pi')
>>> w.get_interact_value()                                                    #␣
→needs sage.symbolic
pi^2
```

class sage.repl.ipython_kernel.widgets.**EvalWidget**(**args*, ***kwds*)

Bases: *TransformWidget*

A mixin class for a widget to evaluate (using `sage_eval()`) the widget value and possibly transform it like *TransformWidget*.

EXAMPLES:

```
sage: from ipywidgets import ToggleButtons
sage: from sage.repl.ipython_kernel.widgets import EvalWidget
sage: class EvalToggleButtons(EvalWidget, ToggleButtons): pass
sage: w = EvalToggleButtons(options=["pi", "e"], transform=lambda x: x+x)
sage: w
EvalToggleButtons(options=('pi', 'e'), value='pi')
sage: w.get_interact_value()                                                  #␣
→needs sage.symbolic
2*pi
```

```
>>> from sage.all import *
>>> from ipywidgets import ToggleButtons
>>> from sage.repl.ipython_kernel.widgets import EvalWidget
>>> class EvalToggleButtons(EvalWidget, ToggleButtons): pass
>>> w = EvalToggleButtons(options=["pi", "e"], transform=lambda x: x+x)
>>> w
EvalToggleButtons(options=('pi', 'e'), value='pi')
>>> w.get_interact_value()                                                    #␣
→needs sage.symbolic
2*pi
```

get_value()

Evaluate the bare widget value using `sage_eval()`.

EXAMPLES:

```
sage: from ipywidgets import Dropdown
sage: from sage.repl.ipython_kernel.widgets import EvalWidget
sage: class EvalDropdown(EvalWidget, Dropdown): pass
sage: w = EvalDropdown(options=["the_answer"], transform=RR)
sage: w
EvalDropdown(options=('the_answer',), value='the_answer')
sage: the_answer = 42
sage: w.get_value()
42
sage: w.get_interact_value()
42.0000000000000
```

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

```
>>> from sage.repl.ipython_kernel.widgets import EvalWidget
>>> class EvalDropdown(EvalWidget, Dropdown): pass
>>> w = EvalDropdown(options=["the_answer"], transform=RR)
>>> w
EvalDropdown(options=('the_answer',), value='the_answer')
>>> the_answer = Integer(42)
>>> w.get_value()
42
>>> w.get_interact_value()
42.0000000000000
```

class sage.repl.ipython_kernel.widgets.**Grid**(***kwargs: Any*)

Bases: *TransformWidget*, `HBox`, `ValueWidget`

A square grid of widgets whose value is a list of lists of the values of the individual widgets.

This is usually created using the `input_grid()` function.

EXAMPLES:

```
sage: from ipywidgets import Text
sage: from sage.repl.ipython_kernel.widgets import Grid
sage: w = Grid(2, 2, lambda i,j: Text(value="%s,%s"%(i,j)))
sage: w
Grid(value=[['0,0', '0,1'], ['1,0', '1,1']],
     children=(Label(value=''),
               VBox(children=(Text(value='0,0'), Text(value='1,0'))),
               VBox(children=(Text(value='0,1'), Text(value='1,1')))))
sage: w.get_interact_value()
[['0,0', '0,1'], ['1,0', '1,1']]
```

```
>>> from sage.all import *
>>> from ipywidgets import Text
>>> from sage.repl.ipython_kernel.widgets import Grid
>>> w = Grid(Integer(2), Integer(2), lambda i,j: Text(value="%s,%s"%(i,j)))
>>> w
Grid(value=[['0,0', '0,1'], ['1,0', '1,1']],
     children=(Label(value=''),
               VBox(children=(Text(value='0,0'), Text(value='1,0'))),
               VBox(children=(Text(value='0,1'), Text(value='1,1')))))
>>> w.get_interact_value()
[['0,0', '0,1'], ['1,0', '1,1']]
```

description

A trait for unicode strings.

value

An instance of a Python list.

class sage.repl.ipython_kernel.widgets.**HTMLText**(***kwargs: Any*)

Bases: `HTMLMath`

An HTML widget whose `description` is always empty.

This is used to display arbitrary HTML text in interacts without a label. The `text_control()` function from SageNB is an alias of *HTMLText*.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.widgets import HTMLText
sage: w = HTMLText("Hello")
sage: w.description
''
sage: w.description = "text"
sage: w.description
''
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.widgets import HTMLText
>>> w = HTMLText("Hello")
>>> w.description
''
>>> w.description = "text"
>>> w.description
''
```

property description

> Always return empty string.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.ipython_kernel.widgets import HTMLText
> sage: w = HTMLText("Hello")
> sage: w.description
> ''
> ```
>
> ```
> >>> from sage.all import *
> >>> from sage.repl.ipython_kernel.widgets import HTMLText
> >>> w = HTMLText("Hello")
> >>> w.description
> ''
> ```

class sage.repl.ipython_kernel.widgets.**SageColorPicker**(***kwargs: Any*)

> Bases: `ColorPicker`
>
> A color picker widget returning a Sage `Color`.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.ipython_kernel.widgets import SageColorPicker
> sage: SageColorPicker()
> SageColorPicker(value='black')
> ```
>
> ```
> >>> from sage.all import *
> >>> from sage.repl.ipython_kernel.widgets import SageColorPicker
> >>> SageColorPicker()
> SageColorPicker(value='black')
> ```

get_interact_value()

> Return a Sage `Color` corresponding to the value of this widget.
>
> EXAMPLES:

```
sage: from sage.repl.ipython_kernel.widgets import SageColorPicker
sage: SageColorPicker().get_interact_value()                                    #␣
↪needs sage.plot
RGB color (0.0, 0.0, 0.0)
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.widgets import SageColorPicker
>>> SageColorPicker().get_interact_value()                                      #␣
↪needs sage.plot
RGB color (0.0, 0.0, 0.0)
```

class sage.repl.ipython_kernel.widgets.**TransformFloatRangeSlider**(***kwargs: Any*)

    Bases: *TransformWidget*, FloatRangeSlider

    An ipywidgets.FloatRangeSlider widget with an optional transformation.

    EXAMPLES:

```
sage: from sage.repl.ipython_kernel.widgets import TransformFloatRangeSlider
sage: w = TransformFloatRangeSlider(min=0, max=100, value=(7, 9),
....:                               transform=lambda x: x[1] - x[0])
sage: w
TransformFloatRangeSlider(value=(7.0, 9.0))
sage: w.get_interact_value()
2.0
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.widgets import TransformFloatRangeSlider
>>> w = TransformFloatRangeSlider(min=Integer(0), max=Integer(100),␣
↪value=(Integer(7), Integer(9)),
...                               transform=lambda x: x[Integer(1)] -␣
↪x[Integer(0)])
>>> w
TransformFloatRangeSlider(value=(7.0, 9.0))
>>> w.get_interact_value()
2.0
```

class sage.repl.ipython_kernel.widgets.**TransformFloatSlider**(***kwargs: Any*)

    Bases: *TransformWidget*, FloatSlider

    A ipywidgets.FloatSlider widget with an optional transformation.

    EXAMPLES:

```
sage: from sage.repl.ipython_kernel.widgets import TransformFloatSlider
sage: w = TransformFloatSlider(min=0, max=100, value=7, transform=lambda x:␣
↪sqrt(x))
sage: w
TransformFloatSlider(value=7.0)
sage: w.get_interact_value()                                                    #␣
↪needs sage.symbolic
2.6457513110645907
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.widgets import TransformFloatSlider
>>> w = TransformFloatSlider(min=Integer(0), max=Integer(100), value=Integer(7),␣
→transform=lambda x: sqrt(x))
>>> w
TransformFloatSlider(value=7.0)
>>> w.get_interact_value()                                                      #␣
→needs sage.symbolic
2.6457513110645907
```

class sage.repl.ipython_kernel.widgets.**TransformIntRangeSlider**(***kwargs: Any*)

Bases: *TransformWidget*, IntRangeSlider

An ipywidgets.IntRangeSlider widget with an optional transformation.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.widgets import TransformIntRangeSlider
sage: w = TransformIntRangeSlider(min=0, max=100, value=(7, 9),
....:                             transform=lambda x: x[1] - x[0])
sage: w
TransformIntRangeSlider(value=(7, 9))
sage: w.get_interact_value()
2
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.widgets import TransformIntRangeSlider
>>> w = TransformIntRangeSlider(min=Integer(0), max=Integer(100),␣
→value=(Integer(7), Integer(9)),
...                              transform=lambda x: x[Integer(1)] - x[Integer(0)])
>>> w
TransformIntRangeSlider(value=(7, 9))
>>> w.get_interact_value()
2
```

class sage.repl.ipython_kernel.widgets.**TransformIntSlider**(***kwargs: Any*)

Bases: *TransformWidget*, IntSlider

An ipywidgets.IntSlider widget with an optional transformation.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.widgets import TransformIntSlider
sage: w = TransformIntSlider(min=0, max=100, value=7, transform=lambda x: x^2)
sage: w
TransformIntSlider(value=7)
sage: w.get_interact_value()
49
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.widgets import TransformIntSlider
>>> w = TransformIntSlider(min=Integer(0), max=Integer(100), value=Integer(7),␣
→transform=lambda x: x**Integer(2))
>>> w
```

(continues on next page)

```
TransformIntSlider(value=7)
>>> w.get_interact_value()
49
```

class sage.repl.ipython_kernel.widgets.**TransformText**(***kwargs: Any*)

> Bases: *TransformWidget*, Text
>
> A ipywidgets.Text widget with an optional transformation.
>
> EXAMPLES:

```
sage: from sage.repl.ipython_kernel.widgets import TransformText
sage: w = TransformText(value='hello', transform=lambda x: x + x)
sage: w
TransformText(value='hello')
sage: w.get_interact_value()
'hellohello'
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.widgets import TransformText
>>> w = TransformText(value='hello', transform=lambda x: x + x)
>>> w
TransformText(value='hello')
>>> w.get_interact_value()
'hellohello'
```

class sage.repl.ipython_kernel.widgets.**TransformTextarea**(***kwargs: Any*)

> Bases: *TransformWidget*, Textarea
>
> A ipywidgets.Textarea widget with an optional transformation.
>
> EXAMPLES:

```
sage: from sage.repl.ipython_kernel.widgets import TransformTextarea
sage: w = TransformTextarea(value='hello', transform=lambda x: x + x)
sage: w
TransformTextarea(value='hello')
sage: w.get_interact_value()
'hellohello'
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.widgets import TransformTextarea
>>> w = TransformTextarea(value='hello', transform=lambda x: x + x)
>>> w
TransformTextarea(value='hello')
>>> w.get_interact_value()
'hellohello'
```

class sage.repl.ipython_kernel.widgets.**TransformWidget**(**args*, ***kwds*)

> Bases: object
>
> A mixin class for a widget to transform the bare widget value for use in interactive functions.
>
> INPUT:

- `transform` – a one-argument function which transforms the value of the widget for use by an interactive function

- other arguments are passed to the base class

EXAMPLES:

```
sage: from ipywidgets import ToggleButtons
sage: from sage.repl.ipython_kernel.widgets import TransformWidget
sage: class TransformToggleButtons(TransformWidget, ToggleButtons): pass
sage: w = TransformToggleButtons(options=["pi", "e"], transform=lambda x: x + x)
sage: w
TransformToggleButtons(options=('pi', 'e'), value='pi')
sage: w.get_interact_value()
'pipi'
```

```
>>> from sage.all import *
>>> from ipywidgets import ToggleButtons
>>> from sage.repl.ipython_kernel.widgets import TransformWidget
>>> class TransformToggleButtons(TransformWidget, ToggleButtons): pass
>>> w = TransformToggleButtons(options=["pi", "e"], transform=lambda x: x + x)
>>> w
TransformToggleButtons(options=('pi', 'e'), value='pi')
>>> w.get_interact_value()
'pipi'
```

get_interact_value()

Return the transformed value of this widget, by calling the `transform` function.

EXAMPLES:

```
sage: from ipywidgets import Checkbox
sage: from sage.repl.ipython_kernel.widgets import TransformWidget
sage: class TransformCheckbox(TransformWidget, Checkbox): pass
sage: w = TransformCheckbox(value=True, transform=int); w
TransformCheckbox(value=True)
sage: w.get_interact_value()
1
```

```
>>> from sage.all import *
>>> from ipywidgets import Checkbox
>>> from sage.repl.ipython_kernel.widgets import TransformWidget
>>> class TransformCheckbox(TransformWidget, Checkbox): pass
>>> w = TransformCheckbox(value=True, transform=int); w
TransformCheckbox(value=True)
>>> w.get_interact_value()
1
```

get_value()

Return `self.value`.

This is meant to be overridden by sub-classes to change the input of the transform function.

EXAMPLES:

```
sage: from ipywidgets import ColorPicker
sage: from sage.repl.ipython_kernel.widgets import TransformWidget
sage: class TransformColorPicker(TransformWidget, ColorPicker): pass
sage: TransformColorPicker(value='red').get_value()
'red'
```

```
>>> from sage.all import *
>>> from ipywidgets import ColorPicker
>>> from sage.repl.ipython_kernel.widgets import TransformWidget
>>> class TransformColorPicker(TransformWidget, ColorPicker): pass
>>> TransformColorPicker(value='red').get_value()
'red'
```

6.6 Installing the SageMath Jupyter Kernel and Extensions

Kernels have to register themselves with Jupyter so that they appear in the Jupyter notebook's kernel drop-down. This is done by *SageKernelSpec*.

> **ⓘ Note**
>
> The doctests in this module run in a temporary directory as the involved directories might be different during runs of the tests and actual installation and because we might be lacking write permission to places such as /usr/share.

class sage.repl.ipython_kernel.install.**SageKernelSpec**(*prefix=None*)

> Bases: object
>
> Utility to manage SageMath kernels and extensions.
>
> INPUT:
>
> - prefix – (default: sys.prefix) directory for the installation prefix
>
> EXAMPLES:

```
sage: from sage.repl.ipython_kernel.install import SageKernelSpec
sage: prefix = tmp_dir()
sage: spec = SageKernelSpec(prefix=prefix)
sage: spec._display_name    # random output
'SageMath 6.9'
sage: spec.kernel_dir == SageKernelSpec(prefix=prefix).kernel_dir
True
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.install import SageKernelSpec
>>> prefix = tmp_dir()
>>> spec = SageKernelSpec(prefix=prefix)
>>> spec._display_name    # random output
'SageMath 6.9'
>>> spec.kernel_dir == SageKernelSpec(prefix=prefix).kernel_dir
True
```

classmethod check()

> Check that the SageMath kernel can be discovered by its name (sagemath).
>
> This method issues a warning if it cannot – either because it is not installed, or it is shadowed by a different kernel of this name, or Jupyter is misconfigured in a different way.
>
> EXAMPLES:

```
sage: from sage.repl.ipython_kernel.install import SageKernelSpec
sage: SageKernelSpec.check()  # random
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.install import SageKernelSpec
>>> SageKernelSpec.check()  # random
```

classmethod identifier()

> Internal identifier for the SageMath kernel.
>
> OUTPUT: the string `'sagemath'`
>
> EXAMPLES:

```
sage: from sage.repl.ipython_kernel.install import SageKernelSpec
sage: SageKernelSpec.identifier()
'sagemath'
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.install import SageKernelSpec
>>> SageKernelSpec.identifier()
'sagemath'
```

kernel_spec()

> Return the kernel spec as Python dictionary.
>
> OUTPUT: a dictionary; see the Jupyter documentation for details
>
> EXAMPLES:

```
sage: from sage.repl.ipython_kernel.install import SageKernelSpec
sage: spec = SageKernelSpec(prefix=tmp_dir())
sage: spec.kernel_spec()
{'argv': ..., 'display_name': 'SageMath ...', 'language': 'sage'}
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.install import SageKernelSpec
>>> spec = SageKernelSpec(prefix=tmp_dir())
>>> spec.kernel_spec()
{'argv': ..., 'display_name': 'SageMath ...', 'language': 'sage'}
```

symlink(*src*, *dst*)

> Symlink `src` to `dst`.
>
> This is not an atomic operation.
>
> Already-existing symlinks will be deleted, already existing non-empty directories will be kept.
>
> EXAMPLES:

```
sage: from sage.repl.ipython_kernel.install import SageKernelSpec
sage: spec = SageKernelSpec(prefix=tmp_dir())
sage: path = tmp_dir()
sage: spec.symlink(os.path.join(path, 'a'), os.path.join(path, 'b'))
sage: os.listdir(path)
['b']
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.install import SageKernelSpec
>>> spec = SageKernelSpec(prefix=tmp_dir())
>>> path = tmp_dir()
>>> spec.symlink(os.path.join(path, 'a'), os.path.join(path, 'b'))
>>> os.listdir(path)
['b']
```

classmethod update(*args*, **kwds*)

   Configure the Jupyter notebook for the SageMath kernel.

   This method does everything necessary to use the SageMath kernel, you should never need to call any of the other methods directly.

   EXAMPLES:

```
sage: from sage.repl.ipython_kernel.install import SageKernelSpec
sage: SageKernelSpec.update(prefix=tmp_dir())
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.install import SageKernelSpec
>>> SageKernelSpec.update(prefix=tmp_dir())
```

use_local_threejs()

   Symlink threejs to the Jupyter notebook.

   EXAMPLES:

```
sage: # needs threejs
sage: from sage.repl.ipython_kernel.install import SageKernelSpec
sage: spec = SageKernelSpec(prefix=tmp_dir())
sage: spec.use_local_threejs()
sage: threejs = os.path.join(spec.nbextensions_dir, 'threejs-sage')
sage: os.path.isdir(threejs)
True
```

```
>>> from sage.all import *
>>> # needs threejs
>>> from sage.repl.ipython_kernel.install import SageKernelSpec
>>> spec = SageKernelSpec(prefix=tmp_dir())
>>> spec.use_local_threejs()
>>> threejs = os.path.join(spec.nbextensions_dir, 'threejs-sage')
>>> os.path.isdir(threejs)
True
```

sage.repl.ipython_kernel.install.**have_prerequisites**(*debug=True*)

   Check that we have all prerequisites to run the Jupyter notebook.

In particular, the Jupyter notebook requires OpenSSL whether or not you are using https. See Issue #17318.

INPUT:

- debug – boolean (default: `True`); whether to print debug information in case that prerequisites are missing

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.install import have_prerequisites
sage: have_prerequisites(debug=False) in [True, False]
True
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.install import have_prerequisites
>>> have_prerequisites(debug=False) in [True, False]
True
```

6.7 The Sage ZMQ Kernel

Version of the Jupyter kernel when running Sage inside the Jupyter notebook or remote Jupyter sessions.

class sage.repl.ipython_kernel.kernel.**SageKernel**(***kwargs: Any*)

Bases: `IPythonKernel`

The Sage Jupyter Kernel.

INPUT:

See the Jupyter documentation.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.kernel import SageKernel
sage: SageKernel.__new__(SageKernel)
<sage.repl.ipython_kernel.kernel.SageKernel object at 0x...>
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.kernel import SageKernel
>>> SageKernel.__new__(SageKernel)
<sage.repl.ipython_kernel.kernel.SageKernel object at 0x...>
```

property banner

The Sage Banner.

The value of this property is displayed in the Jupyter notebook.

OUTPUT: string

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.kernel import SageKernel
sage: sk = SageKernel.__new__(SageKernel)
sage: print(sk.banner)
┌...SageMath version...
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.kernel import SageKernel
>>> sk = SageKernel.__new__(SageKernel)
>>> print(sk.banner)
⌐...SageMath version...
```

property help_links

Help in the Jupyter Notebook.

OUTPUT: see the Jupyter documentation

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.kernel import SageKernel
sage: sk = SageKernel.__new__(SageKernel)
sage: sk.help_links
[{'text': 'Sage Documentation',
  'url': '.../html/en/index.html'},
 ...]
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_kernel.kernel import SageKernel
>>> sk = SageKernel.__new__(SageKernel)
>>> sk.help_links
[{'text': 'Sage Documentation',
  'url': '.../html/en/index.html'},
 ...]
```

implementation: str = 'sage'

implementation_version: str = '10.6'

pre_handler_hook()

Restore the signal handlers to their default values at Sage startup, saving the old handler at the `saved_sig-int_handler` attribute. This is needed because Jupyter needs to change the `SIGINT` handler.

See Issue #19135.

shell_class

A trait whose value must be a subclass of a specified class.

class sage.repl.ipython_kernel.kernel.**SageZMQInteractiveShell**(**_kwargs: Any_)

Bases: _SageNotebookInteractiveShell_, ZMQInteractiveShell

6.8 Tests for the IPython integration

First, test the pinfo magic for Python code. This is what IPython calls when you ask for the single-questionmark help, like foo?

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell(u'from sage.repl.ipython_tests import dummy')
sage: shell.run_cell(u'%pinfo dummy')
Signature:      dummy(argument, optional=None)
```

(continues on next page)

```
Docstring:
   Dummy Docstring Title.

   Dummy docstring explanation.

   INPUT:

   ... "argument" -- anything; dummy argument

   ... "optional" -- anything (optional); dummy optional

   EXAMPLES...

...
Init docstring: ...ee help(type(...)) for...signature...
File:           .../sage/repl/ipython_tests.py
Type:           function
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell()
>>> shell.run_cell(u'from sage.repl.ipython_tests import dummy')
>>> shell.run_cell(u'%pinfo dummy')
Signature:      dummy(argument, optional=None)
Docstring:
   Dummy Docstring Title.
<BLANKLINE>
   Dummy docstring explanation.
<BLANKLINE>
   INPUT:
<BLANKLINE>
   ... "argument" -- anything; dummy argument
<BLANKLINE>
   ... "optional" -- anything (optional); dummy optional
<BLANKLINE>
   EXAMPLES...
<BLANKLINE>
...
Init docstring: ...ee help(type(...)) for...signature...
File:           .../sage/repl/ipython_tests.py
Type:           function
```

Next, test the pinfo magic for Cython code:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell(u'from sage.tests.stl_vector import stl_int_vector')
sage: shell.run_cell(u'%pinfo stl_int_vector')
...
   Example class wrapping an STL vector.

   EXAMPLES...
```

```
...
Init docstring: ...ee help(type(...)) for...signature...
File:           .../sage/tests/stl_vector.pyx
Type:           type
...
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell()
>>> shell.run_cell(u'from sage.tests.stl_vector import stl_int_vector')
>>> shell.run_cell(u'%pinfo stl_int_vector')
...
   Example class wrapping an STL vector.
<BLANKLINE>
   EXAMPLES...
<BLANKLINE>
...
Init docstring: ...ee help(type(...)) for...signature...
File:           .../sage/tests/stl_vector.pyx
Type:           type
...
```

Next, test the `pinfo` magic for R interface code, see Issue #26906:

```
sage: from sage.repl.interpreter import get_test_shell    # optional - rpy2
sage: shell = get_test_shell()                            # optional - rpy2
sage: shell.run_cell(u'%pinfo r.lm')                      # optional - rpy2
Signature:      r.lm(...*args, **kwds)
...
String form:    lm
File:           .../sage/interfaces/r.py
Docstring:...
title
*****

Fitting Linear Models
...
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell    # optional - rpy2
>>> shell = get_test_shell()                            # optional - rpy2
>>> shell.run_cell(u'%pinfo r.lm')                      # optional - rpy2
Signature:      r.lm(...*args, **kwds)
...
String form:    lm
File:           .../sage/interfaces/r.py
Docstring:...
title
*****
<BLANKLINE>
Fitting Linear Models
```

```
...
```

Next, test the pinfo2 magic for Python code. This is what IPython calls when you ask for the double-questionmark help, like foo??

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell(u'from sage.repl.ipython_tests import dummy')
sage: shell.run_cell(u'%pinfo2 dummy')
Signature: dummy(argument, optional=None)
...
Source:
def dummy(argument, optional=None):
    """
    Dummy Docstring Title.

    Dummy docstring explanation.

    INPUT:

    - ``argument`` -- anything; dummy argument

    - ``optional`` -- anything (optional); dummy optional

    EXAMPLES::

    ...
    """
    return 'Source code would be here'
File:      .../sage/repl/ipython_tests.py
Type:      function
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell()
>>> shell.run_cell(u'from sage.repl.ipython_tests import dummy')
>>> shell.run_cell(u'%pinfo2 dummy')
Signature: dummy(argument, optional=None)
...
Source:
def dummy(argument, optional=None):
    """
    Dummy Docstring Title.
<BLANKLINE>
    Dummy docstring explanation.
<BLANKLINE>
    INPUT:
<BLANKLINE>
    - ``argument`` -- anything; dummy argument
<BLANKLINE>
    - ``optional`` -- anything (optional); dummy optional
<BLANKLINE>
```

```
    EXAMPLES::
<BLANKLINE>
    ...
    """
    return 'Source code would be here'
File:      .../sage/repl/ipython_tests.py
Type:      function
```

Next, test the pinfo2 magic for Cython code:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell(u'from sage.tests.stl_vector import stl_int_vector')
sage: shell.run_cell(u'%pinfo2 stl_int_vector')
...
cdef class stl_int_vector(SageObject):
    """
    Example class wrapping an STL vector.

    EXAMPLES::


...
    """

    cdef vector[int] *data
    cdef string *name

    def __cinit__(self):
        """
        The Cython constructor.

        EXAMPLES::


...
File:   .../sage/tests/stl_vector.pyx
Type:   type
...
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell()
>>> shell.run_cell(u'from sage.tests.stl_vector import stl_int_vector')
>>> shell.run_cell(u'%pinfo2 stl_int_vector')
...
cdef class stl_int_vector(SageObject):
    """
    Example class wrapping an STL vector.
<BLANKLINE>
    EXAMPLES::
<BLANKLINE>
...
    """
```

```
<BLANKLINE>
    cdef vector[int] *data
    cdef string *name
<BLANKLINE>
    def __cinit__(self):
        """
        The Cython constructor.
<BLANKLINE>
        EXAMPLES::
<BLANKLINE>
...
File:   .../sage/tests/stl_vector.pyx
Type:   type
...
```

Next, test the `pinfo2` magic for R interface code, see Issue #26906:

```
sage: from sage.repl.interpreter import get_test_shell   # optional - rpy2
sage: shell = get_test_shell()                           # optional - rpy2
sage: shell.run_cell(u'%pinfo2 r.lm')                    # optional - rpy2
Signature:      r.lm(...*args, **kwds)
...
String form:    lm
File:           .../sage/interfaces/r.py
Source:
function (formula, data, subset, weights, na.action, method = "qr",
...
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell   # optional - rpy2
>>> shell = get_test_shell()                           # optional - rpy2
>>> shell.run_cell(u'%pinfo2 r.lm')                    # optional - rpy2
Signature:      r.lm(...*args, **kwds)
...
String form:    lm
File:           .../sage/interfaces/r.py
Source:
function (formula, data, subset, weights, na.action, method = "qr",
...
```

Test that there are no warnings being ignored internally:

```
sage: import warnings
sage: warnings.simplefilter('error');  get_test_shell()
<sage.repl.interpreter.SageTestShell object at 0x...>
```

```
>>> from sage.all import *
>>> import warnings
>>> warnings.simplefilter('error');  get_test_shell()
<sage.repl.interpreter.SageTestShell object at 0x...>
```

sage.repl.ipython_tests.**dummy**(*argument*, *optional=None*)

    Dummy Docstring Title.

Dummy docstring explanation.

INPUT:

- `argument` – anything; dummy argument

- `optional` – anything (optional); dummy optional

EXAMPLES:

```
sage: from sage.repl.ipython_tests import dummy
sage: dummy(1)
'Source code would be here'
```

```
>>> from sage.all import *
>>> from sage.repl.ipython_tests import dummy
>>> dummy(Integer(1))
'Source code would be here'
```

6.9 HTML Generator for JSmol

This is all an evil iframe hack to get JSmol to display 3-d graphics while separating JSmol's j2s machinery from your actual web page.

There are some caveats for how to load JSmol, in particular it cannot just load its code from a `file://` uri. To use a html file generated by this module, you need

- A web server,

- The JSmol directory tree must be served by your web server,

- The output of *JSMolHtml.inner_html()* or *JSMolHtml.outer_html()* must be served by the same web server.

See https://github.com/phetsims/molecule-polarity/issues/6 for a discussion of loading JSMol.

class sage.repl.display.jsmol_iframe.**JSMolHtml**(*jmol*, *path_to_jsmol=None*, *width='100%'*, *height='100%'*)

Bases: `SageObject`

INPUT:

- `jmol` – 3-d graphics or *sage.repl.rich_output.output_graphics3d.OutputSceneJmol* instance. The 3-d scene to show.

- `path_to_jsmol` – string (default: `'/nbextensions/jupyter-jsmol/jsmol'`). The path (relative or absolute) where `JSmol.min.js` is served on the web server.

- `width` – integer or string (default: `'100%'`). The width of the JSmol applet using CSS dimensions.

- `height` – integer or string (default: `'100%'`). The height of the JSmol applet using CSS dimensions.

EXAMPLES:

```
sage: from sage.repl.display.jsmol_iframe import JSMolHtml
sage: JSMolHtml(sphere(), width=500, height=300)                          #
↪needs sage.plot
JSmol Window 500x300
```

```
>>> from sage.all import *
>>> from sage.repl.display.jsmol_iframe import JSMolHtml
>>> JSMolHtml(sphere(), width=Integer(500), height=Integer(300))        ↪
↪            # needs sage.plot
JSmol Window 500x300
```

`iframe()`

> Return HTML iframe.
>
> OUTPUT: string
>
> EXAMPLES:

```
sage: from sage.repl.display.jsmol_iframe import JSMolHtml
sage: from sage.repl.rich_output.output_graphics3d import OutputSceneJmol
sage: jmol = JSMolHtml(OutputSceneJmol.example())
sage: print(jmol.iframe())
<iframe srcdoc="
...
</iframe>
```

```
>>> from sage.all import *
>>> from sage.repl.display.jsmol_iframe import JSMolHtml
>>> from sage.repl.rich_output.output_graphics3d import OutputSceneJmol
>>> jmol = JSMolHtml(OutputSceneJmol.example())
>>> print(jmol.iframe())
<iframe srcdoc="
...
</iframe>
```

`inner_html()`

> Return a HTML document containing a JSmol applet.
>
> EXAMPLES:

```
sage: from sage.repl.display.jsmol_iframe import JSMolHtml
sage: from sage.repl.rich_output.output_graphics3d import OutputSceneJmol
sage: jmol = JSMolHtml(OutputSceneJmol.example(), width=500, height=300)
sage: print(jmol.inner_html())
<html>
<head>
  <style>
    * {
      margin: 0;
      padding: 0;
        ...
</html>
```

```
>>> from sage.all import *
>>> from sage.repl.display.jsmol_iframe import JSMolHtml
>>> from sage.repl.rich_output.output_graphics3d import OutputSceneJmol
>>> jmol = JSMolHtml(OutputSceneJmol.example(), width=Integer(500),↪
↪height=Integer(300))
>>> print(jmol.inner_html())
```

```
<html>
<head>
  <style>
    * {
      margin: 0;
      padding: 0;
        ...
</html>
```

js_script()

The *script()* as Javascript string.

Since the many shortcomings of Javascript include multi-line strings, this actually returns Javascript code to reassemble the script from a list of strings.

OUTPUT:

String. Javascript code that evaluates to *script()* as Javascript string.

EXAMPLES:

```
sage: from sage.repl.display.jsmol_iframe import JSMolHtml
sage: from sage.repl.rich_output.output_graphics3d import OutputSceneJmol
sage: jsmol = JSMolHtml(OutputSceneJmol.example(), width=500, height=300)
sage: print(jsmol.js_script())
[
  'data "model list"',
  ...
  'isosurface fullylit; pmesh o* fullylit; set antialiasdisplay on;',
].join('\n');
```

```
>>> from sage.all import *
>>> from sage.repl.display.jsmol_iframe import JSMolHtml
>>> from sage.repl.rich_output.output_graphics3d import OutputSceneJmol
>>> jsmol = JSMolHtml(OutputSceneJmol.example(), width=Integer(500),␣
→height=Integer(300))
>>> print(jsmol.js_script())
[
  'data "model list"',
  ...
  'isosurface fullylit; pmesh o* fullylit; set antialiasdisplay on;',
].join('\n');
```

outer_html()

Return a HTML document containing an iframe with a JSmol applet.

OUTPUT: string

EXAMPLES:

```
sage: from sage.repl.display.jsmol_iframe import JSMolHtml
sage: from sage.repl.rich_output.output_graphics3d import OutputSceneJmol
sage: jmol = JSMolHtml(OutputSceneJmol.example(), width=500, height=300)
sage: print(jmol.outer_html())
<html>
```

```
<head>
  <title>JSmol 3D Scene</title>
</head>
</body>

<iframe srcdoc="
      ...
</html>
```

```
>>> from sage.all import *
>>> from sage.repl.display.jsmol_iframe import JSMolHtml
>>> from sage.repl.rich_output.output_graphics3d import OutputSceneJmol
>>> jmol = JSMolHtml(OutputSceneJmol.example(), width=Integer(500),␣
↪height=Integer(300))
>>> print(jmol.outer_html())
<html>
<head>
  <title>JSmol 3D Scene</title>
</head>
</body>
<BLANKLINE>
<iframe srcdoc="
      ...
</html>
```

script()

> Return the JMol script file.
>
> This method extracts the Jmol script from the Jmol spt file (a zip archive) and inlines meshes.
>
> OUTPUT: string
>
> EXAMPLES:

```
sage: from sage.repl.display.jsmol_iframe import JSMolHtml
sage: from sage.repl.rich_output.output_graphics3d import OutputSceneJmol
sage: jsmol = JSMolHtml(OutputSceneJmol.example(), width=500, height=300)
sage: jsmol.script()
'data "model list"\n10\nempt...aliasdisplay on;\n'
```

```
>>> from sage.all import *
>>> from sage.repl.display.jsmol_iframe import JSMolHtml
>>> from sage.repl.rich_output.output_graphics3d import OutputSceneJmol
>>> jsmol = JSMolHtml(OutputSceneJmol.example(), width=Integer(500),␣
↪height=Integer(300))
>>> jsmol.script()
'data "model list"\n10\nempt...aliasdisplay on;\n'
```

6.10 Sage Wrapper for Bitmap Images

Some computations in Sage return bitmap images, for example matrices can be turned into bitmaps directly. Note that this is different from all plotting functionality, the latter can equally produce vector graphics. This module is about bitmaps

only, and a shallow wrapper around `PIL.Image`. The only difference is that *Image* is displayed as graphics by the Sage if the UI can.

EXAMPLES:

```
sage: from sage.repl.image import Image
sage: img = Image('RGB', (256, 256), 'white')
sage: pixels = img.pixels()
sage: for x in range(img.width()):
....:     for y in range(img.height()):
....:         pixels[x, y] = (x, y, 100)
sage: img
256x256px 24-bit RGB image
sage: type(img)
<class 'sage.repl.image.Image'>
```

```
>>> from sage.all import *
>>> from sage.repl.image import Image
>>> img = Image('RGB', (Integer(256), Integer(256)), 'white')
>>> pixels = img.pixels()
>>> for x in range(img.width()):
...     for y in range(img.height()):
...         pixels[x, y] = (x, y, Integer(100))
>>> img
256x256px 24-bit RGB image
>>> type(img)
<class 'sage.repl.image.Image'>
```

class `sage.repl.image.`**`Image`**(*mode*, *size*, *color='white'*)

Bases: `SageObject`

Create a new image with the given mode and size.

INPUT:

- `mode` – string. The mode to use for the new image. Valid options are:

    - `'1'` (1-bit pixels, black and white, stored with one pixel per byte)

    - `'L'` (8-bit pixels, black and white)

    - `'P'` (8-bit pixels, mapped to any other mode using a color palette)

    - `'RGB'` (3x8-bit pixels, true color)

    - `'RGBA'` (4x8-bit pixels, true color with transparency mask)

    - `'CMYK'` (4x8-bit pixels, color separation)

    - `'YCbCr'` (3x8-bit pixels, color video format)

    - `'LAB'` (3x8-bit pixels, the L*a*b color space)

    - `'HSV'` (3x8-bit pixels, Hue, Saturation, Value color space)

    - `'I'` (32-bit signed integer pixels)

    - `'F'` (32-bit floating point pixels)

- `size` – 2-tuple, containing (width, height) in pixels

- `color` – string, numeric or tuple of numeric. What colour to use for the image. Default is black. If given, this should be a a tuple with one value per band. When creating RGB images, you can also use colour strings as supported by the ImageColor module. If the colour is None, the image is not initialised.

OUTPUT: a new *Image* object

EXAMPLES:

```
sage: from sage.repl.image import Image
sage: Image('P', (16, 16), 13)
16x16px 8-bit Color image
```

```
>>> from sage.all import *
>>> from sage.repl.image import Image
>>> Image('P', (Integer(16), Integer(16)), Integer(13))
16x16px 8-bit Color image
```

height()

> Return the vertical dimension in pixels.
>
> OUTPUT: integer
>
> EXAMPLES:

```
sage: from sage.repl.image import Image
sage: img = Image('1', (12, 34), 'white')
sage: img.width()
12
sage: img.height()
34
```

```
>>> from sage.all import *
>>> from sage.repl.image import Image
>>> img = Image('1', (Integer(12), Integer(34)), 'white')
>>> img.width()
12
>>> img.height()
34
```

mode()

> Return the color mode.
>
> OUTPUT: string; as given when constructing the image
>
> EXAMPLES:

```
sage: from sage.repl.image import Image
sage: img = Image('YCbCr', (16, 16), 'white')
sage: img.mode()
'YCbCr'
```

```
>>> from sage.all import *
>>> from sage.repl.image import Image
>>> img = Image('YCbCr', (Integer(16), Integer(16)), 'white')
>>> img.mode()
'YCbCr'
```

property pil

Access the wrapped PIL(low) Image.

OUTPUT: the underlying `PIL.Image.Image object`

EXAMPLES:

```
sage: from sage.repl.image import Image
sage: img = Image('RGB', (16, 16), 'white')
sage: img.pil
<PIL.Image.Image image mode=RGB size=16x16...>
```

```
>>> from sage.all import *
>>> from sage.repl.image import Image
>>> img = Image('RGB', (Integer(16), Integer(16)), 'white')
>>> img.pil
<PIL.Image.Image image mode=RGB size=16x16...>
```

pixels()

Return the pixel map.

OUTPUT:

The PIL PixelAccess object that allows you to get/set the pixel data.

EXAMPLES:

```
sage: from sage.repl.image import Image
sage: img = Image('RGB', (16, 16), 'white')
sage: img.pixels()
<PixelAccess object at 0x...>
```

```
>>> from sage.all import *
>>> from sage.repl.image import Image
>>> img = Image('RGB', (Integer(16), Integer(16)), 'white')
>>> img.pixels()
<PixelAccess object at 0x...>
```

save(*filename*)

Save the bitmap image.

INPUT:

- `filename` – string; the filename to save as. The given extension automatically determines the image file type

EXAMPLES:

```
sage: from sage.repl.image import Image
sage: img = Image('P', (12, 34), 13)
sage: filename = tmp_filename(ext='.png')
sage: img.save(filename)
sage: with open(filename, 'rb') as f:
....:     f.read(4) == b'\x89PNG'
True
```

```
>>> from sage.all import *
>>> from sage.repl.image import Image
>>> img = Image('P', (Integer(12), Integer(34)), Integer(13))
>>> filename = tmp_filename(ext='.png')
>>> img.save(filename)
>>> with open(filename, 'rb') as f:
...     f.read(Integer(4)) == b'\x89PNG'
True
```

show()

Show this image 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.

OUTPUT:

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

EXAMPLES:

```
sage: from sage.repl.image import Image
sage: img = Image('1', (12, 34), 'white')
sage: img.show()
```

```
>>> from sage.all import *
>>> from sage.repl.image import Image
>>> img = Image('1', (Integer(12), Integer(34)), 'white')
>>> img.show()
```

width()

Return the horizontal dimension in pixels.

OUTPUT: integer

EXAMPLES:

```
sage: from sage.repl.image import Image
sage: img = Image('1', (12, 34), 'white')
sage: img.width()
12
sage: img.height()
34
```

```
>>> from sage.all import *
>>> from sage.repl.image import Image
>>> img = Image('1', (Integer(12), Integer(34)), 'white')
>>> img.width()
12
>>> img.height()
34
```

6.11 The Sage Input Hook

This lets us perform actions while IPython is sitting at the terminal input prompt. We use it to reload attached files if they have changed.

sage.repl.inputhook.**install**()

> Install the Sage input hook.
>
> EXAMPLES:
>
> Make sure ipython is running so we really test this function:

```
sage: from sage.repl.interpreter import get_test_shell
sage: get_test_shell()
<sage.repl.interpreter.SageTestShell object at ...>
```

```
>>> from sage.all import *
>>> from sage.repl.interpreter import get_test_shell
>>> get_test_shell()
<sage.repl.interpreter.SageTestShell object at ...>
```

> Run the function twice, to check it is idempotent (see Issue #35235):

```
sage: from sage.repl.inputhook import install
sage: install()
sage: install()
```

```
>>> from sage.all import *
>>> from sage.repl.inputhook import install
>>> install()
>>> install()
```

sage.repl.inputhook.**sage_inputhook**(*context*)

sage.repl.inputhook.**uninstall**()

> Uninstall the Sage input hook.
>
> EXAMPLES:

```
sage: from sage.repl.inputhook import uninstall
sage: uninstall()
```

```
>>> from sage.all import *
>>> from sage.repl.inputhook import uninstall
>>> uninstall()
```

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

INDEX

Non-alphabetical