Sage's Doctesting Framework

Release 10.5.rc0

The Sage Development Team

Nov 16, 2024

CONTENTS

CHAPTER

ONE

CLASSES INVOLVED IN DOCTESTING

This module controls the various classes involved in doctesting.

AUTHORS:

- David Roe (2012-03-27) – initial version, based on Robert Bradshaw's code.

class sage.doctest.control.**DocTestController**(*options*, *args*)

Bases: SageObject

This class controls doctesting of files.

After creating it with appropriate options, call the *run()* method to run the doctests.

add_files()

Check for the flags '–all' and '–new'.

For each one present, this function adds the appropriate directories and files to the todo list.

EXAMPLES:

```
sage: from sage.doctest.control import (DocTestDefaults,
....:                                   DocTestController)
sage: from sage.env import SAGE_SRC
sage: import tempfile
sage: with tempfile.NamedTemporaryFile() as f:
....:     DD = DocTestDefaults(all=True, logfile=f.name)
....:     DC = DocTestController(DD, [])
....:     DC.add_files()
Doctesting ...
sage: os.path.join(SAGE_SRC, 'sage') in DC.files
True
```

```
sage: DD = DocTestDefaults(new = True)
sage: DC = DocTestController(DD, [])
sage: DC.add_files()
Doctesting ...
```

cleanup(*final=True*)

Run cleanup activities after actually running doctests.

In particular, saves the stats to disk and closes the logfile.

INPUT:

- `final` – whether to close the logfile

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults, DocTestController
sage: from sage.env import SAGE_SRC
sage: import os
sage: dirname = os.path.join(SAGE_SRC, 'sage', 'rings', 'all.py')
sage: DD = DocTestDefaults()

sage: DC = DocTestController(DD, [dirname])
sage: DC.expand_files_into_sources()
sage: DC.sources.sort(key=lambda s:s.basename)

sage: for i, source in enumerate(DC.sources):
....:     DC.stats[source.basename] = {'walltime': 0.1r * (i+1)}
....:

sage: DC.run()
Running doctests with ID ...
Doctesting 1 file.
sage -t .../rings/all.py
    [... tests, ...s wall]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: ... seconds
    cpu time: ... seconds
    cumulative wall time: ... seconds
Features detected...
0
sage: DC.cleanup()
```

create_run_id()

Create the run id.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults, DocTestController
sage: DC = DocTestController(DocTestDefaults(), [])
sage: DC.create_run_id()
Running doctests with ID ...
```

expand_files_into_sources()

Expand `self.files`, which may include directories, into a list of `sage.doctest.FileDocTestSource`

This function also handles the optional command line option.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults, DocTestController
sage: from sage.env import SAGE_SRC
sage: import os
sage: dirname = os.path.join(SAGE_SRC, 'sage', 'doctest')
sage: DD = DocTestDefaults(optional='all')
sage: DC = DocTestController(DD, [dirname])
sage: DC.expand_files_into_sources()
sage: len(DC.sources)
15
sage: DC.sources[0].options.optional
True
```

```
sage: DD = DocTestDefaults(optional='magma,guava')
sage: DC = DocTestController(DD, [dirname])
sage: DC.expand_files_into_sources()
sage: all(t in DC.sources[0].options.optional for t in ['magma','guava'])
True
```

We check that files are skipped appropriately:

```
sage: dirname = tmp_dir()
sage: filename = os.path.join(dirname, 'not_tested.py')
sage: with open(filename, 'w') as f:
....:     _ = f.write("#"*80 + "\n\n\n\n## nodoctest\n    sage: 1+1\n    4")
sage: DC = DocTestController(DD, [dirname])
sage: DC.expand_files_into_sources()
sage: DC.sources
[]
```

The directory `sage/doctest/tests` contains `nodoctest.py` but the files should still be tested when that directory is explicitly given (as opposed to being recursed into):

```
sage: DC = DocTestController(DD, [os.path.join(SAGE_SRC, 'sage', 'doctest',
↪'tests')])
sage: DC.expand_files_into_sources()
sage: len(DC.sources) >= 10
True
```

filter_sources()

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults, DocTestController
sage: from sage.env import SAGE_SRC
sage: import os
sage: dirname = os.path.join(SAGE_SRC, 'sage', 'doctest')
sage: DD = DocTestDefaults(failed=True)
sage: DC = DocTestController(DD, [dirname])
sage: DC.expand_files_into_sources()
sage: for i, source in enumerate(DC.sources):
....:     DC.stats[source.basename] = {'walltime': 0.1r * (i+1)}
sage: DC.stats['sage.doctest.control'] = {'failed': True, 'walltime': 1.0r}
sage: DC.filter_sources()
Only doctesting files that failed last test.
sage: len(DC.sources)
1
```

load_baseline_stats(*filename*)

Load baseline stats.

This must be a JSON file in the same format that *load_stats()* expects.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults, DocTestController
sage: DC = DocTestController(DocTestDefaults(), [])
sage: import json
sage: filename = tmp_filename()
sage: with open(filename, 'w') as stats_file:
....:     json.dump({'sage.doctest.control':{'failed':True}}, stats_file)
```

(continues on next page)

(continued from previous page)

```
sage: DC.load_baseline_stats(filename)
sage: DC.baseline_stats['sage.doctest.control']
{'failed': True}
```

If the file doesn't exist, nothing happens. If there is an error, print a message. In any case, leave the stats alone:

```
sage: d = tmp_dir()
sage: DC.load_baseline_stats(os.path.join(d))  # Cannot read a directory
Error loading baseline stats from ...
sage: DC.load_baseline_stats(os.path.join(d, "no_such_file"))
sage: DC.baseline_stats['sage.doctest.control']
{'failed': True}
```

load_environment()

Return the module that provides the global environment.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults, DocTestController
sage: DC = DocTestController(DocTestDefaults(), [])
sage: 'BipartiteGraph' in DC.load_environment().__dict__
True
sage: DC = DocTestController(DocTestDefaults(environment='sage.doctest.all'),␣
↪[])
sage: 'BipartiteGraph' in  DC.load_environment().__dict__
False
sage: 'run_doctests' in DC.load_environment().__dict__
True
```

load_stats(*filename*)

Load stats from the most recent run(s).

Stats are stored as a JSON file, and include information on which files failed tests and the walltime used for execution of the doctests.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults, DocTestController
sage: DC = DocTestController(DocTestDefaults(), [])
sage: import json
sage: filename = tmp_filename()
sage: with open(filename, 'w') as stats_file:
....:     json.dump({'sage.doctest.control': {'walltime': 1.0r}}, stats_file)
sage: DC.load_stats(filename)
sage: DC.stats['sage.doctest.control']
{'walltime': 1.0}
```

If the file doesn't exist, nothing happens. If there is an error, print a message. In any case, leave the stats alone:

```
sage: d = tmp_dir()
sage: DC.load_stats(os.path.join(d))  # Cannot read a directory
Error loading stats from ...
sage: DC.load_stats(os.path.join(d, "no_such_file"))
sage: DC.stats['sage.doctest.control']
{'walltime': 1.0}
```

log(*s*, *end='\n'*)

Log the string `s + end` (where `end` is a newline by default) to the logfile and print it to the standard output.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults, DocTestController
sage: DD = DocTestDefaults(logfile=tmp_filename())
sage: DC = DocTestController(DD, [])
sage: DC.log("hello world")
hello world
sage: DC.logfile.close()
sage: with open(DD.logfile) as f:
....:     print(f.read())
hello world
```

In serial mode, check that logging works even if `stdout` is redirected:

```
sage: DD = DocTestDefaults(logfile=tmp_filename(), serial=True)
sage: DC = DocTestController(DD, [])
sage: from sage.doctest.forker import SageSpoofInOut
sage: with open(os.devnull, 'w') as devnull:
....:     S = SageSpoofInOut(devnull)
....:     S.start_spoofing()
....:     DC.log("hello world")
....:     S.stop_spoofing()
hello world
sage: DC.logfile.close()
sage: with open(DD.logfile) as f:
....:     print(f.read())
hello world
```

Check that no duplicate logs appear, even when forking (Issue #15244):

```
sage: DD = DocTestDefaults(logfile=tmp_filename())
sage: DC = DocTestController(DD, [])
sage: DC.log("hello world")
hello world
sage: if os.fork() == 0:
....:     DC.logfile.close()
....:     os._exit(0)
sage: DC.logfile.close()
sage: with open(DD.logfile) as f:
....:     print(f.read())
hello world
```

run()

This function is called after initialization to set up and run all doctests.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults, DocTestController
sage: from sage.env import SAGE_SRC
sage: import os
sage: DD = DocTestDefaults()
sage: filename = os.path.join(SAGE_SRC, "sage", "sets", "non_negative_
↪integers.py")
sage: DC = DocTestController(DD, [filename])
sage: DC.run()
```

(continues on next page)

(continued from previous page)

```
Running doctests with ID ...
Doctesting 1 file.
sage -t .../sage/sets/non_negative_integers.py
    [... tests, ...s wall]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: ... seconds
    cpu time: ... seconds
    cumulative wall time: ... seconds
Features detected...
0
```

We check that Issue #25378 is fixed (testing external packages while providing a logfile does not raise a ValueError: I/O operation on closed file):

```
sage: logfile = tmp_filename(ext='.log')
sage: DD = DocTestDefaults(optional=set(['sage', 'external']),␣
↪logfile=logfile)
sage: filename = tmp_filename(ext='.py')
sage: DC = DocTestController(DD, [filename])
sage: DC.run()
Running doctests with ID ...
Using --optional=external,sage
Features to be detected: ...
Doctesting 1 file.
sage -t ....py
    [0 tests, ...s wall]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: ... seconds
    cpu time: ... seconds
    cumulative wall time: ... seconds
Features detected...
0
```

We test the `--hide` option (Issue #34185):

```
sage: from sage.doctest.control import test_hide
sage: filename = tmp_filename(ext='.py')
sage: with open(filename, 'w') as f:
....:     f.write(test_hide)
....:     f.close()
714
sage: DF = DocTestDefaults(hide='buckygen,all')
sage: DC = DocTestController(DF, [filename])
sage: DC.run()
Running doctests with ID ...
Using --optional=sage...
Features to be detected: ...
Doctesting 1 file.
sage -t ....py
    [4 tests, ...s wall]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
Total time for all tests: ... seconds
    cpu time: ... seconds
    cumulative wall time: ... seconds
Features detected...
0

sage: DF = DocTestDefaults(hide='benzene,optional')
sage: DC = DocTestController(DF, [filename])
sage: DC.run()
Running doctests with ID ...
Using --optional=sage
Features to be detected: ...
Doctesting 1 file.
sage -t ....py
    [4 tests, ...s wall]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: ... seconds
    cpu time: ... seconds
    cumulative wall time: ... seconds
Features detected...
0
```

Test *Features that have been hidden* message:

```
sage: DC.run()                                # optional - meataxe
Running doctests with ID ...
Using --optional=sage
Features to be detected: ...
Doctesting 1 file.
sage -t ....py
    [4 tests, ...s wall]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: ... seconds
    cpu time: ... seconds
    cumulative wall time: ... seconds
Features detected...
Features that have been hidden: ...meataxe...
0
```

run_doctests()

Actually run the doctests.

This function is called by *run()*.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults, DocTestController
sage: from sage.env import SAGE_SRC
sage: import os
sage: dirname = os.path.join(SAGE_SRC, 'sage', 'rings', 'homset.py')
sage: DD = DocTestDefaults()
sage: DC = DocTestController(DD, [dirname])
sage: DC.expand_files_into_sources()
```

(continues on next page)

(continued from previous page)

```
sage: DC.run_doctests()
Doctesting 1 file.
sage -t .../sage/rings/homset.py
    [... tests, ...s wall]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: ... seconds
    cpu time: ... seconds
    cumulative wall time: ... seconds...
```

run_val_gdb (*testing=False*)

Spawns a subprocess to run tests under the control of gdb, lldb, or valgrind.

INPUT:

- `testing` – boolean (default: `False`); if `True` then the command to be run will be printed rather than a subprocess started

EXAMPLES:

Note that the command lines include unexpanded environment variables. It is safer to let the shell expand them than to expand them here and risk insufficient quoting.

```
sage: from sage.doctest.control import DocTestDefaults, DocTestController
sage: DD = DocTestDefaults(gdb=True)
sage: DC = DocTestController(DD, ["hello_world.py"])
sage: DC.run_val_gdb(testing=True)
exec gdb --eval-command="run" --args ...python... -m sage.doctest --serial...␣
↪--timeout=0... hello_world.py
```

```
sage: DD = DocTestDefaults(valgrind=True, optional='all', timeout=172800)
sage: DC = DocTestController(DD, ["hello_world.py"])
sage: DC.run_val_gdb(testing=True)
exec valgrind --tool=memcheck --leak-resolution=high --leak-check=full --num-
↪callers=25 --suppressions=.../valgrind/pyalloc.supp --suppressions=.../
↪valgrind/sage.supp --suppressions=.../valgrind/sage-additional.supp --
↪suppressions=.../valgrind/valgrind-python.supp  --log-file=.../valgrind/
↪sage-memcheck.%p ...python... -m sage.doctest --serial... --timeout=172800..
↪. --optional=all hello_world.py
```

save_stats (*filename*)

Save stats from the most recent run as a JSON file.

WARNING: This function overwrites the file.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults, DocTestController
sage: DC = DocTestController(DocTestDefaults(), [])
sage: DC.stats['sage.doctest.control'] = {'walltime': 1.0r}
sage: filename = tmp_filename()
sage: DC.save_stats(filename)
sage: import json
sage: with open(filename) as f:
....:     D = json.load(f)
sage: D['sage.doctest.control']
{'walltime': 1.0}
```

second_on_modern_computer()

Return the wall time equivalent of a second on a modern computer.

OUTPUT:

Float. The wall time on your computer that would be equivalent to one second on a modern computer. Unless you have kick-ass hardware this should always be >= 1.0. This raises a `RuntimeError` if there are no stored timings to use as benchmark.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults, DocTestController
sage: DC = DocTestController(DocTestDefaults(), [])
sage: DC.second_on_modern_computer()    # not tested
```

sort_sources()

This function sorts the sources so that slower doctests are run first.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults, DocTestController
sage: from sage.env import SAGE_SRC
sage: import os
sage: dirname = os.path.join(SAGE_SRC, 'sage', 'doctest')
sage: DD = DocTestDefaults(nthreads=2)
sage: DC = DocTestController(DD, [dirname])
sage: DC.expand_files_into_sources()
sage: DC.sources.sort(key=lambda s:s.basename)
sage: for i, source in enumerate(DC.sources):
....:     DC.stats[source.basename] = {'walltime': 0.1r * (i+1)}
sage: DC.sort_sources()
Sorting sources by runtime so that slower doctests are run first....
sage: print("\n".join(source.basename for source in DC.sources))
sage.doctest.util
sage.doctest.test
sage.doctest.sources
sage.doctest.rif_tol
sage.doctest.reporting
sage.doctest.parsing_test
sage.doctest.parsing
sage.doctest.marked_output
sage.doctest.forker
sage.doctest.fixtures
sage.doctest.external
sage.doctest.control
sage.doctest.check_tolerance
sage.doctest.all
sage.doctest
```

source_baseline(*source*)

Return the `baseline_stats` value of `source`.

INPUT:

- `source` – a `DocTestSource` instance

OUTPUT: a dictionary

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults, DocTestController
sage: filename = sage.doctest.util.__file__
sage: DD = DocTestDefaults()
sage: DC = DocTestController(DD, [filename])
sage: DC.expand_files_into_sources()
sage: DC.source_baseline(DC.sources[0])
{}
```

class sage.doctest.control.**DocTestDefaults**(*runtest_default=False*, ***kwds*)

Bases: SageObject

This class is used for doctesting the Sage doctest module.

INPUT:

- runtest_default – (boolean, default False); if True, fills in attribute to be the same as the defaults defined in sage-runtests. If False, change defaults in a few places for use in doctests of the doctester, which is mostly to make doctesting more predictable.
- **kwds – attributes to override defaults

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: D = DocTestDefaults(); D
DocTestDefaults()
sage: D.timeout
-1
```

Keyword arguments become attributes:

```
sage: D = DocTestDefaults(timeout=100); D
DocTestDefaults(timeout=100)
sage: D.timeout
100
```

The defaults for sage-runtests:

```
sage: D = DocTestDefaults(runtest_default=True); D
DocTestDefaults(abspath=False, file_iterations=0, global_iterations=0,
                optional='sage,optional', random_seed=None,
                stats_path='.../timings2.json')
```

class sage.doctest.control.**Logger**(**files*)

Bases: object

File-like object which implements writing to multiple files at once.

EXAMPLES:

```
sage: from sage.doctest.control import Logger
sage: with open(tmp_filename(), "w+") as t:
....:     L = Logger(sys.stdout, t)
....:     _ = L.write("hello world\n")
....:     _ = t.seek(0)
....:     t.read()
hello world
'hello world\n'
```

flush()

Flush all files.

write(*x*)

Write x to all files.

sage.doctest.control.**run_doctests**(*module*, *options=None*)

Run the doctests in a given file.

INPUT:

- module – a Sage module, a string, or a list of such
- options – a DocTestDefaults object or None

EXAMPLES:

```
sage: run_doctests(sage.rings.all)
Running doctests with ID ...
Doctesting 1 file.
sage -t .../sage/rings/all.py
    [... tests, ...s wall]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: ... seconds
    cpu time: ... seconds
    cumulative wall time: ... seconds
Features detected...
```

sage.doctest.control.**skipdir**(*dirname*)

Return True if and only if the directory dirname should not be doctested.

EXAMPLES:

```
sage: from sage.doctest.control import skipdir
sage: skipdir(sage.env.SAGE_SRC)
False
sage: skipdir(os.path.join(sage.env.SAGE_SRC, "sage", "doctest", "tests"))
True
```

sage.doctest.control.**skipfile**(*filename*, *tested_optional_tags*, *if_installed*, *log=False*)

Return True if and only if the file filename should not be doctested.

INPUT:

- filename – name of a file
- tested_optional_tags – list or tuple or set of optional tags to test, or False (no optional test) or True (all optional tests)
- if_installed – boolean (default: False); whether to skip Python/Cython files that are not installed as modules
- log – function to call with log messages, or None

If filename contains a line of the form "# sage.doctest: optional - xyz"), then this will return False if "xyz" is in tested_optional_tags. Otherwise, it returns the matching tag ("optional - xyz").

EXAMPLES:

```
sage: from sage.doctest.control import skipfile
sage: skipfile("skipme.c")
True
sage: filename = tmp_filename(ext='.pyx')
sage: skipfile(filename)
False
sage: with open(filename, "w") as f:
....:     _ = f.write("# nodoctest")
sage: skipfile(filename)
True
sage: with open(filename, "w") as f:
....:     _ = f.write("# sage.doctest: "    # broken in two source lines to avoid␣
↪the pattern
....:                 "optional - xyz")     # of relint (multiline_doctest_
↪comment)
sage: skipfile(filename, False)
'optional - xyz'
sage: bool(skipfile(filename, False))
True
sage: skipfile(filename, ['abc'])
'optional - xyz'
sage: skipfile(filename, ['abc', 'xyz'])
False
sage: skipfile(filename, True)
False
```

CHAPTER

TWO

CLASSES FOR SOURCES OF DOCTESTS

This module defines various classes for sources from which doctests originate, such as files, functions or database entries.

AUTHORS:

- David Roe (2012-03-27) – initial version, based on Robert Bradshaw's code.

class sage.doctest.sources.**DictAsObject**(*attrs*)

Bases: dict

A simple subclass of dict that inserts the items from the initializing dictionary into attributes.

EXAMPLES:

```
sage: from sage.doctest.sources import DictAsObject
sage: D = DictAsObject({'a':2})
sage: D.a
2
```

class sage.doctest.sources.**DocTestSource**(*options*)

Bases: object

This class provides a common base class for different sources of doctests.

INPUT:

- options – a *sage.doctest.control.DocTestDefaults* instance or equivalent

file_optional_tags()

Return the set of tags that should apply to all doctests in this source.

This default implementation just returns the empty set.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import StringDocTestSource, PythonSource
sage: from sage.structure.dynamic_class import dynamic_class
sage: s = "'''\n    sage: 2 + 2\n    4\n'''"
sage: PythonStringSource = dynamic_class('PythonStringSource',␣
↪(StringDocTestSource, PythonSource))
sage: PSS = PythonStringSource('<runtime>', s, DocTestDefaults(), 'runtime')
sage: PSS.file_optional_tags
set()
```

class sage.doctest.sources.**FileDocTestSource**(*path*, *options*)

Bases: *DocTestSource*

This class creates doctests from a file.

INPUT:

- `path` – string; the filename
- `options` – a *sage.doctest.control.DocTestDefaults* instance or equivalent

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: filename = sage.doctest.sources.__file__
sage: FDS = FileDocTestSource(filename, DocTestDefaults())
sage: FDS.basename
'sage.doctest.sources'
```

basename()

The basename of this file source, e.g. `sage.doctest.sources`.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: filename = os.path.join(SAGE_SRC,'sage','rings','integer.pyx')
sage: FDS = FileDocTestSource(filename, DocTestDefaults())
sage: FDS.basename
'sage.rings.integer'
```

create_doctests(*namespace*)

Return a list of doctests for this file.

INPUT:

- `namespace` – dictionary or *sage.doctest.util.RecordingDict*

OUTPUT:

- `doctests` – list of doctests defined in this file
- `extras` – dictionary

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: filename = sage.doctest.sources.__file__
sage: FDS = FileDocTestSource(filename, DocTestDefaults())
sage: doctests, extras = FDS.create_doctests(globals())
sage: len(doctests)
43
sage: extras['tab']
False
```

We give a self referential example:

```
sage: doctests[20].name
'sage.doctest.sources.FileDocTestSource.create_doctests'
sage: doctests[20].examples[8].source
'doctests[Integer(20)].examples[Integer(8)].source\n'
```

file_optional_tags()

Return the set of tags that should apply to all doctests in this source.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: filename = sage.repl.user_globals.__file__
sage: FDS = FileDocTestSource(filename, DocTestDefaults())
sage: FDS.file_optional_tags
{'sage.modules': None}
```

in_lib()

Whether this file is to be treated as a module in a Python package.

Such files aren't loaded before running tests.

This uses `is_package_or_sage_namespace_package_dir()` but can be overridden via *DocTestDefaults*.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: import os
sage: filename = os.path.join(SAGE_SRC, 'sage', 'rings', 'integer.pyx')
sage: FDS = FileDocTestSource(filename, DocTestDefaults())
sage: FDS.in_lib
True
sage: filename = os.path.join(SAGE_SRC, 'sage', 'doctest', 'tests', 'abort.rst
↪')
sage: FDS = FileDocTestSource(filename, DocTestDefaults())
sage: FDS.in_lib
False
```

You can override the default:

```
sage: FDS = FileDocTestSource("hello_world.py", DocTestDefaults())
sage: FDS.in_lib
False
sage: FDS = FileDocTestSource("hello_world.py", DocTestDefaults(force_
↪lib=True))
sage: FDS.in_lib
True
```

printpath()

Whether the path is printed absolutely or relatively depends on an option.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: import os
sage: filename = os.path.realpath(sage.doctest.sources.__file__)
sage: root = os.path.join(os.path.dirname(filename), '..')
sage: cwd = os.getcwd()
sage: os.chdir(root)
sage: FDS = FileDocTestSource(filename, DocTestDefaults(randorder=0,
....:                                                   abspath=False))
```

(continues on next page)

(continued from previous page)

```
sage: FDS.printpath
'doctest/sources.py'
sage: FDS = FileDocTestSource(filename, DocTestDefaults(randorder=0,
....:                                                   abspath=True))
sage: FDS.printpath
'.../sage/doctest/sources.py'
sage: os.chdir(cwd)
```

class sage.doctest.sources.**PythonSource**

Bases: *SourceLanguage*

This class defines the functions needed for the extraction of doctests from python sources.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: filename = sage.doctest.sources.__file__
sage: FDS = FileDocTestSource(filename, DocTestDefaults())
sage: type(FDS)
<class 'sage.doctest.sources.PythonFileSource'>
```

ending_docstring(*line*)

Determines whether the input line ends a docstring.

INPUT:

- `line` – string, one line of an input file

OUTPUT: an object that, when evaluated in a boolean context, gives `True` or `False` depending on whether the input line marks the end of a docstring

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.doctest.util import NestedName
sage: filename = sage.doctest.sources.__file__
sage: FDS = FileDocTestSource(filename, DocTestDefaults())
sage: FDS._init()
sage: FDS.quotetype = "'''"
sage: FDS.ending_docstring("'''")
<...Match object...>
sage: FDS.ending_docstring('\"\"\"')
```

start_finish_can_overlap = False

starting_docstring(*line*)

Determines whether the input line starts a docstring.

If the input line does start a docstring (a triple quote), then this function updates `self.qualified_name`.

INPUT:

- `line` – string; one line of an input file

OUTPUT: either `None` or a Match object

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.doctest.util import NestedName
sage: filename = sage.doctest.sources.__file__
sage: FDS = FileDocTestSource(filename, DocTestDefaults())
sage: FDS._init()
sage: FDS.starting_docstring("r'''")
<...Match object...>
sage: FDS.ending_docstring("'''")
<...Match object...>
sage: FDS.qualified_name = NestedName(FDS.basename)
sage: FDS.starting_docstring("class MyClass():")
sage: FDS.starting_docstring("    def hello_world(self):")
sage: FDS.starting_docstring("        '''")
<...Match object...>
sage: FDS.qualified_name
sage.doctest.sources.MyClass.hello_world
sage: FDS.ending_docstring("    '''")
<...Match object...>
sage: FDS.starting_docstring("class NewClass():")
sage: FDS.starting_docstring("    '''")
<...Match object...>
sage: FDS.ending_docstring("    '''")
<...Match object...>
sage: FDS.qualified_name
sage.doctest.sources.NewClass
sage: FDS.starting_docstring("print(")
sage: FDS.starting_docstring("    '''Not a docstring")
sage: FDS.starting_docstring("    ''')")
sage: FDS.starting_docstring("def foo():")
sage: FDS.starting_docstring("    '''This is a docstring'''")
<...Match object...>
```

class sage.doctest.sources.**RestSource**

Bases: *SourceLanguage*

This class defines the functions needed for the extraction of doctests from ReST sources.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: filename = "sage_doc.rst"
sage: FDS = FileDocTestSource(filename, DocTestDefaults())
sage: type(FDS)
<class 'sage.doctest.sources.RestFileSource'>
```

ending_docstring(*line*)

When the indentation level drops below the initial level the block ends.

INPUT:

- `line` – string; one line of an input file

OUTPUT: boolean; whether the verbatim block is ending

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: filename = "sage_doc.rst"
sage: FDS = FileDocTestSource(filename, DocTestDefaults())
sage: FDS._init()
sage: FDS.starting_docstring("Hello world::")
True
sage: FDS.ending_docstring("    sage: 2 + 2")
False
sage: FDS.ending_docstring("    4")
False
sage: FDS.ending_docstring("We are now done")
True
```

parse_docstring(*docstring*, *namespace*, *start*)

Return a list of doctest defined in this docstring.

Code blocks in a REST file can contain python functions with their own docstrings in addition to in-line doctests. We want to include the tests from these inner docstrings, but Python's doctesting module has a problem if we just pass on the whole block, since it expects to get just a docstring, not the Python code as well.

Our solution is to create a new doctest source from this code block and append the doctests created from that source. We then replace the occurrences of "sage:" and ">>>" occurring inside a triple quote with "safe:" so that the doctest module doesn't treat them as tests.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.doctest.parsing import SageDocTestParser
sage: from sage.doctest.util import NestedName
sage: filename = "sage_doc.rst"
sage: FDS = FileDocTestSource(filename, DocTestDefaults())
sage: FDS.parser = SageDocTestParser(set(['sage']))
sage: FDS.qualified_name = NestedName('sage_doc')
sage: s = "Some text::\n\n    def example_python_function(a, \
....:      b):\n        '''\n        Brief description \
....:      of function.\n\n        EXAMPLES::\n\n            \
....:      sage: test1()\n              sage: test2()\n        \
....:      '''\n        return a + b\n\n    sage: test3()\n\nMore \
....:      ReST documentation."
sage: tests = FDS.parse_docstring(s, {}, 100)
sage: len(tests)
2
sage: for ex in tests[0].examples:
....:     print(ex.sage_source)
test3()
sage: for ex in tests[1].examples:
....:     print(ex.sage_source)
test1()
test2()
sig_on_count() # check sig_on/off pairings (virtual doctest)
```

start_finish_can_overlap = True

starting_docstring(*line*)

A line ending with a double colon starts a verbatim block in a ReST file, as does a line containing `..`

CODE-BLOCK:: language.

This function also determines whether the docstring block should be joined with the previous one, or should be skipped.

INPUT:

- line – string; one line of an input file

OUTPUT: either None or a Match object

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: filename = "sage_doc.rst"
sage: FDS = FileDocTestSource(filename, DocTestDefaults())
sage: FDS._init()
sage: FDS.starting_docstring("Hello world::")
True
sage: FDS.ending_docstring("    sage: 2 + 2")
False
sage: FDS.ending_docstring("    4")
False
sage: FDS.ending_docstring("We are now done")
True
sage: FDS.starting_docstring(".. link")
sage: FDS.starting_docstring("::")
True
sage: FDS.linking
True
```

class sage.doctest.sources.**SourceLanguage**

Bases: object

An abstract class for functions that depend on the programming language of a doctest source.

Currently supported languages include Python, ReST and LaTeX.

parse_docstring(*docstring*, *namespace*, *start*)

Return a list of doctest defined in this docstring.

This function is called by DocTestSource._process_doc(). The default implementation, defined here, is to use the *sage.doctest.parsing.SageDocTestParser* attached to this source to get doctests from the docstring.

INPUT:

- docstring – string containing documentation and tests
- namespace – dictionary or *sage.doctest.util.RecordingDict*
- start – integer; one less than the starting line number

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.doctest.parsing import SageDocTestParser
sage: from sage.doctest.util import NestedName
sage: filename = sage.doctest.util.__file__
sage: FDS = FileDocTestSource(filename, DocTestDefaults())
```

(continues on next page)

(continued from previous page)

```
sage: doctests, _ = FDS.create_doctests({})
sage: for dt in doctests:
....:     FDS.qualified_name = dt.name
....:     dt.examples = dt.examples[:-1] # strip off the sig_on() test
....:     assert(FDS.parse_docstring(dt.docstring,{},dt.lineno-1)[0] == dt)
```

class sage.doctest.sources.**StringDocTestSource**(*basename*, *source*, *options*, *printpath*, *lineno_shift=0*)

Bases: *DocTestSource*

This class creates doctests from a string.

INPUT:

- `basename` – string such as 'sage.doctests.sources', going into the names of created doctests and examples
- `source` – string, giving the source code to be parsed for doctests
- `options` – a *sage.doctest.control.DocTestDefaults* or equivalent
- `printpath` – string, to be used in place of a filename when doctest failures are displayed
- `lineno_shift` – integer (default: 0) by which to shift the line numbers of all doctests defined in this string

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import StringDocTestSource, PythonSource
sage: from sage.structure.dynamic_class import dynamic_class
sage: s = "'''\n    sage: 2 + 2\n    4\n'''"
sage: PythonStringSource = dynamic_class('PythonStringSource',
↪(StringDocTestSource, PythonSource))
sage: PSS = PythonStringSource('<runtime>', s, DocTestDefaults(), 'runtime')
sage: dt, extras = PSS.create_doctests({})
sage: len(dt)
1
sage: extras['tab']
[]
sage: extras['line_number']
False

sage: s = "'''\n\tsage: 2 + 2\n\t4\n'''"
sage: PSS = PythonStringSource('<runtime>', s, DocTestDefaults(), 'runtime')
sage: dt, extras = PSS.create_doctests({})
sage: extras['tab']
['2', '3']

sage: s = "'''\n    sage: import warnings; warnings.warn('foo')\n    doctest:1:␣
↪UserWarning: foo \n'''"
sage: PSS = PythonStringSource('<runtime>', s, DocTestDefaults(), 'runtime')
sage: dt, extras = PSS.create_doctests({})
sage: extras['line_number']
True
```

create_doctests(*namespace*)

Create doctests from this string.

INPUT:

- `namespace` – dictionary or *sage.doctest.util.RecordingDict*

OUTPUT:

- `doctests` – list of doctests defined by this string
- `tab_locations` – either `False` or a list of linenumbers on which tabs appear

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import StringDocTestSource, PythonSource
sage: from sage.structure.dynamic_class import dynamic_class
sage: s = "'''\n    sage: 2 + 2\n    4\n'''"
sage: PythonStringSource = dynamic_class('PythonStringSource',
↪(StringDocTestSource, PythonSource))
sage: PSS = PythonStringSource('<runtime>', s, DocTestDefaults(), 'runtime')
sage: dt, tabs = PSS.create_doctests({})
sage: for t in dt:
....:     print("{} {}".format(t.name, t.examples[0].sage_source))
<runtime> 2 + 2
```

class sage.doctest.sources.**TexSource**

Bases: *SourceLanguage*

This class defines the functions needed for the extraction of doctests from a LaTeX source.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: filename = "sage_paper.tex"
sage: FDS = FileDocTestSource(filename, DocTestDefaults())
sage: type(FDS)
<class 'sage.doctest.sources.TexFileSource'>
```

ending_docstring(*line*, *check_skip=True*)

Determines whether the input line ends a docstring.

Docstring blocks in tex files are defined by verbatim or lstlisting environments, and can be linked together by adding %link immediately after the end{verbatim} or end{lstlisting}.

Within a verbatim (or lstlisting) block, you can tell Sage not to process the rest of the block by including a %skip line.

INPUT:

- `line` – string, one line of an input file
- `check_skip` – boolean (default: `True`); used internally in `starting_docstring`

OUTPUT: boolean; whether the input line marks the end of a docstring (verbatim block)

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: filename = "sage_paper.tex"
sage: FDS = FileDocTestSource(filename, DocTestDefaults())
sage: FDS._init()
sage: FDS.ending_docstring(r"\end{verbatim}")
True
sage: FDS.ending_docstring(r"\end{lstlisting}")
```

(continues on next page)

(continued from previous page)

```
True
sage: FDS.linking
False
```

Use %link to link with the next verbatim block:

```
sage: FDS.ending_docstring(r"\end{verbatim}%link")
True
sage: FDS.linking
True
```

%skip also ends a docstring block:

```
sage: FDS.ending_docstring("%skip")
True
```

start_finish_can_overlap = False

starting_docstring(*line*)

Determines whether the input line starts a docstring.

Docstring blocks in tex files are defined by verbatim or lstlisting environments, and can be linked together by adding %link immediately after the end{verbatim} or end{lstlisting}.

Within a verbatim (or lstlisting) block, you can tell Sage not to process the rest of the block by including a %skip line.

INPUT:

- `line` – string, one line of an input file

OUTPUT: boolean; whether the input line marks the start of a docstring (verbatim block)

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: filename = "sage_paper.tex"
sage: FDS = FileDocTestSource(filename, DocTestDefaults())
sage: FDS._init()
```

We start docstrings with begin{verbatim} or begin{lstlisting}:

```
sage: FDS.starting_docstring(r"\begin{verbatim}")
True
sage: FDS.starting_docstring(r"\begin{lstlisting}")
True
sage: FDS.skipping
False
sage: FDS.ending_docstring("sage: 2+2")
False
sage: FDS.ending_docstring("4")
False
```

To start ignoring the rest of the verbatim block, use %skip:

```
sage: FDS.ending_docstring("%skip")
True
```

(continues on next page)

(continued from previous page)

```
sage: FDS.skipping
True
sage: FDS.starting_docstring("sage: raise RuntimeError")
False
```

You can even pretend to start another verbatim block while skipping:

```
sage: FDS.starting_docstring(r"\begin{verbatim}")
False
sage: FDS.skipping
True
```

To stop skipping end the verbatim block:

```
sage: FDS.starting_docstring(r"\end{verbatim} %link")
False
sage: FDS.skipping
False
```

Linking works even when the block was ended while skipping:

```
sage: FDS.linking
True
sage: FDS.starting_docstring(r"\begin{verbatim}")
True
```

sage.doctest.sources.**get_basename**(*path*)

This function returns the basename of the given path, e.g. `sage.doctest.sources` or `doc.ru.tutorial.tour_advanced`.

EXAMPLES:

```
sage: from sage.doctest.sources import get_basename
sage: import os
sage: get_basename(sage.doctest.sources.__file__)
'sage.doctest.sources'
sage: get_basename(os.path.join(sage.structure.__path__[0], 'element.pxd'))
'sage.structure.element.pxd'
```

CHAPTER

THREE

PROCESSES FOR RUNNING DOCTESTS

This module controls the processes started by Sage that actually run the doctests.

EXAMPLES:

The following examples are used in doctesting this file:

```
sage: doctest_var = 42; doctest_var^2
1764
sage: R.<a> = ZZ[]
sage: a + doctest_var
a + 42
```

AUTHORS:

- David Roe (2012-03-27) – initial version, based on Robert Bradshaw's code.
- Jeroen Demeyer (2013 and 2015) – major improvements to forking and logging

class sage.doctest.forker.**DocTestDispatcher**(*controller*)

Bases: SageObject

Create parallel *DocTestWorker* processes and dispatches doctesting tasks.

dispatch()

Run the doctests for the controller's specified sources, by calling *parallel_dispatch()* or *serial_dispatch()* according to the `--serial` option.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestController, DocTestDefaults
sage: from sage.doctest.forker import DocTestDispatcher
sage: from sage.doctest.reporting import DocTestReporter
sage: from sage.doctest.util import Timer
sage: import os
sage: freehom = os.path.join(SAGE_SRC, 'sage', 'modules', 'free_module_
↪homspace.py')
sage: bigo = os.path.join(SAGE_SRC, 'sage', 'rings', 'big_oh.py')
sage: DC = DocTestController(DocTestDefaults(), [freehom, bigo])
sage: DC.expand_files_into_sources()
sage: DD = DocTestDispatcher(DC)
sage: DR = DocTestReporter(DC)
sage: DC.reporter = DR
sage: DC.dispatcher = DD
sage: DC.timer = Timer().start()
sage: DD.dispatch()
sage -t .../sage/modules/free_module_homspace.py
```

(continues on next page)

(continued from previous page)

```
    [... tests, ...s wall]
sage -t .../sage/rings/big_oh.py
    [... tests, ...s wall]
```

parallel_dispatch()

Run the doctests from the controller's specified sources in parallel.

This creates *DocTestWorker* subprocesses, while the master process checks for timeouts and collects and displays the results.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestController, DocTestDefaults
sage: from sage.doctest.forker import DocTestDispatcher
sage: from sage.doctest.reporting import DocTestReporter
sage: from sage.doctest.util import Timer
sage: import os
sage: crem = os.path.join(SAGE_SRC, 'sage', 'databases', 'cremona.py')
sage: bigo = os.path.join(SAGE_SRC, 'sage', 'rings', 'big_oh.py')
sage: DC = DocTestController(DocTestDefaults(), [crem, bigo])
sage: DC.expand_files_into_sources()
sage: DD = DocTestDispatcher(DC)
sage: DR = DocTestReporter(DC)
sage: DC.reporter = DR
sage: DC.dispatcher = DD
sage: DC.timer = Timer().start()
sage: DD.parallel_dispatch()
sage -t .../databases/cremona.py
    [... tests, ...s wall]
sage -t .../rings/big_oh.py
    [... tests, ...s wall]
```

If the `exitfirst=True` option is given, the results for a failing module will be immediately printed and any other ongoing tests canceled:

```
sage: from tempfile import NamedTemporaryFile as NTF
sage: with NTF(suffix='.py', mode='w+t') as f1, \
....:      NTF(suffix='.py', mode='w+t') as f2:
....:     _ = f1.write("'''\nsage: import time; time.sleep(60)\n'''")
....:     f1.flush()
....:     _ = f2.write("'''\nsage: True\nFalse\n'''")
....:     f2.flush()
....:     DC = DocTestController(DocTestDefaults(exitfirst=True,
....:                                            nthreads=2),
....:                            [f1.name, f2.name])
....:     DC.expand_files_into_sources()
....:     DD = DocTestDispatcher(DC)
....:     DR = DocTestReporter(DC)
....:     DC.reporter = DR
....:     DC.dispatcher = DD
....:     DC.timer = Timer().start()
....:     DD.parallel_dispatch()
sage -t ...
**********************************************************************
File "...", line 2, in ...
Failed example:
    True
```

(continues on next page)

(continued from previous page)

```
Expected:
    False
Got:
    True
**********************************************************************
1 item had failures:
   1 of   1 in ...
    [1 test, 1 failure, ...s wall]
Killing test ...
```

serial_dispatch()

Run the doctests from the controller's specified sources in series.

There is no graceful handling for signals, no possibility of interrupting tests and no timeout.

EXAMPLES:

```
sage: from sage.doctest.control import DocTestController, DocTestDefaults
sage: from sage.doctest.forker import DocTestDispatcher
sage: from sage.doctest.reporting import DocTestReporter
sage: from sage.doctest.util import Timer
sage: import os
sage: homset = os.path.join(SAGE_SRC, 'sage', 'rings', 'homset.py')
sage: ideal = os.path.join(SAGE_SRC, 'sage', 'rings', 'ideal.py')
sage: DC = DocTestController(DocTestDefaults(), [homset, ideal])
sage: DC.expand_files_into_sources()
sage: DD = DocTestDispatcher(DC)
sage: DR = DocTestReporter(DC)
sage: DC.reporter = DR
sage: DC.dispatcher = DD
sage: DC.timer = Timer().start()
sage: DD.serial_dispatch()
sage -t .../rings/homset.py
    [... tests, ...s wall]
sage -t .../rings/ideal.py
    [... tests, ...s wall]
```

class sage.doctest.forker.**DocTestTask**(*source*)

Bases: object

This class encapsulates the tests from a single source.

This class does not insulate from problems in the source (e.g. entering an infinite loop or causing a segfault), that has to be dealt with at a higher level.

INPUT:

- source – a *sage.doctest.sources.DocTestSource* instance
- verbose – boolean, controls reporting of progress by doctest.DocTestRunner

EXAMPLES:

```
sage: from sage.doctest.forker import DocTestTask
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.doctest.control import DocTestDefaults, DocTestController
sage: import os
sage: filename = sage.doctest.sources.__file__
sage: DD = DocTestDefaults()
```

(continues on next page)

(continued from previous page)

```
sage: FDS = FileDocTestSource(filename, DD)
sage: DTT = DocTestTask(FDS)
sage: DC = DocTestController(DD,[filename])
sage: ntests, results = DTT(options=DD)
sage: ntests >= 300 or ntests
True
sage: sorted(results.keys())
['cputime', 'err', 'failures', 'optionals', 'tests', 'walltime', 'walltime_skips']
```

class sage.doctest.forker.**DocTestWorker**(*source*, *options*, *funclist=[]*, *baseline=None*)

Bases: `Process`

The DocTestWorker process runs one *DocTestTask* for a given source. It returns messages about doctest failures (or all tests if verbose doctesting) through a pipe and returns results through a `multiprocessing.Queue` instance (both these are created in the *start()* method).

It runs the task in its own process-group, such that killing the process group kills this process together with its child processes.

The class has additional methods and attributes for bookkeeping by the master process. Except in *run()*, nothing from this class should be accessed by the child process.

INPUT:

- `source` – a `DocTestSource` instance
- `options` – an object representing doctest options
- `funclist` – list of callables to be called at the start of the child process
- `baseline` – dictionary, the `baseline_stats` value

EXAMPLES:

```
sage: from sage.doctest.forker import DocTestWorker, DocTestTask
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.doctest.reporting import DocTestReporter
sage: from sage.doctest.control import DocTestController, DocTestDefaults
sage: filename = sage.doctest.util.__file__
sage: DD = DocTestDefaults()
sage: FDS = FileDocTestSource(filename, DD)
sage: W = DocTestWorker(FDS, DD)
sage: W.start()
sage: DC = DocTestController(DD, filename)
sage: reporter = DocTestReporter(DC)
sage: W.join()  # Wait for worker to finish
sage: result = W.result_queue.get()
sage: reporter.report(FDS, False, W.exitcode, result, "")
    [... tests, ...s wall]
```

kill()

Kill this worker. Return `True` if the signal(s) are sent successfully or `False` if the worker process no longer exists.

This method is only called if there is something wrong with the worker. Under normal circumstances, the worker is supposed to exit by itself after finishing.

The first time this is called, use `SIGQUIT`. This will trigger the cysignals `SIGQUIT` handler and try to print an enhanced traceback.

Subsequent times, use `SIGKILL`. Also close the message pipe if it was still open.

EXAMPLES:

```
sage: import time
sage: from sage.doctest.forker import DocTestWorker, DocTestTask
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.doctest.reporting import DocTestReporter
sage: from sage.doctest.control import DocTestController, DocTestDefaults
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','tests','99seconds.rst
↪')
sage: DD = DocTestDefaults()
sage: FDS = FileDocTestSource(filename, DD)
```

We set up the worker to start by blocking `SIGQUIT`, such that killing will fail initially:

```
sage: from cysignals.pselect import PSelecter
sage: import signal
sage: def block_hup():
....:     # We never __exit__()
....:     PSelecter([signal.SIGQUIT]).__enter__()
sage: W = DocTestWorker(FDS, DD, [block_hup])
sage: W.start()
sage: W.killed
False
sage: W.kill()
True
sage: W.killed
True
sage: time.sleep(float(0.2))  # Worker doesn't die
sage: W.kill()          # Worker dies now
True
sage: time.sleep(1)
sage: W.is_alive()
False
```

read_messages()

In the master process, read from the pipe and store the data read in the `messages` attribute.

Note

This function may need to be called multiple times in order to read all of the messages.

EXAMPLES:

```
sage: from sage.doctest.forker import DocTestWorker, DocTestTask
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.doctest.reporting import DocTestReporter
sage: from sage.doctest.control import DocTestController, DocTestDefaults
sage: filename = sage.doctest.util.__file__
sage: DD = DocTestDefaults(verbose=True,nthreads=2)
sage: FDS = FileDocTestSource(filename, DD)
sage: W = DocTestWorker(FDS, DD)
sage: W.start()
sage: while W.rmessages is not None:
....:     W.read_messages()
```

(continues on next page)

(continued from previous page)

```
sage: W.join()
sage: len(W.messages) > 0
True
```

run()

Run the *DocTestTask* under its own PGID.

save_result_output()

Annotate `self` with `self.result` (the result read through the `result_queue` and with `self.output`, the complete contents of `self.outtmpfile`. Then close the Queue and `self.outtmpfile`.

EXAMPLES:

```
sage: from sage.doctest.forker import DocTestWorker, DocTestTask
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.doctest.reporting import DocTestReporter
sage: from sage.doctest.control import DocTestController, DocTestDefaults
sage: filename = sage.doctest.util.__file__
sage: DD = DocTestDefaults()
sage: FDS = FileDocTestSource(filename, DD)
sage: W = DocTestWorker(FDS, DD)
sage: W.start()
sage: W.join()
sage: W.save_result_output()
sage: sorted(W.result[1].keys())
['cputime', 'err', 'failures', 'optionals', 'tests', 'walltime', 'walltime_
↪skips']
sage: len(W.output) > 0
True
```

Note

This method is called from the parent process, not from the subprocess.

start()

Start the worker and close the writing end of the message pipe.

class sage.doctest.forker.**SageDocTestRunner**(**args*, ***kwds*)

Bases: DocTestRunner

A customized version of DocTestRunner that tracks dependencies of doctests.

INPUT:

- `stdout` – an open file to restore for debugging
- `checker` – None, or an instance of `doctest.OutputChecker`
- `verbose` – boolean, determines whether verbose printing is enabled
- `optionflags` – controls the comparison with the expected output. See `testmod` for more information
- `baseline` – dictionary, the `baseline_stats` value

EXAMPLES:

```
sage: from sage.doctest.parsing import SageOutputChecker
sage: from sage.doctest.forker import SageDocTestRunner
sage: from sage.doctest.control import DocTestDefaults; DD = DocTestDefaults()
sage: import doctest, sys, os
sage: DTR = SageDocTestRunner(SageOutputChecker(), verbose=False, sage_options=DD,
↪ optionflags=doctest.NORMALIZE_WHITESPACE|doctest.ELLIPSIS)
sage: DTR
<sage.doctest.forker.SageDocTestRunner object at ...>
```

compile_and_execute(*example*, *compiler*, *globs*)

Run the given example, recording dependencies.

Rather than using a basic dictionary, Sage's doctest runner uses a *sage.doctest.util.RecordingDict*, which records every time a value is set or retrieved. Executing the given code with this recording dictionary as the namespace allows Sage to track dependencies between doctest lines. For example, in the following two lines

```
sage: R.<x> = ZZ[]
sage: f = x^2 + 1
```

the recording dictionary records that the second line depends on the first since the first INSERTS `x` into the global namespace and the second line RETRIEVES `x` from the global namespace.

INPUT:

- `example` – a `doctest.Example` instance
- `compiler` – a callable that, applied to example, produces a code object
- `globs` – dictionary in which to execute the code

OUTPUT: the output of the compiled code snippet

EXAMPLES:

```
sage: from sage.doctest.parsing import SageOutputChecker
sage: from sage.doctest.forker import SageDocTestRunner
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.doctest.util import RecordingDict
sage: from sage.doctest.control import DocTestDefaults; DD = DocTestDefaults()
sage: import doctest, sys, os, hashlib
sage: DTR = SageDocTestRunner(SageOutputChecker(), verbose=False, sage_
↪options=DD,
....:           optionflags=doctest.NORMALIZE_WHITESPACE|doctest.ELLIPSIS)
sage: DTR.running_doctest_digest = hashlib.md5()
sage: filename = sage.doctest.forker.__file__
sage: FDS = FileDocTestSource(filename, DD)
sage: globs = RecordingDict(globals())
sage: 'doctest_var' in globs
False
sage: doctests, extras = FDS.create_doctests(globs)
sage: ex0 = doctests[0].examples[0]
sage: flags = 32768 if sys.version_info.minor < 8 else 524288
sage: def compiler(ex):
....:     return compile(ex.source, '<doctest sage.doctest.forker[0]>',
....:                    'single', flags, 1)
sage: DTR.compile_and_execute(ex0, compiler, globs)
1764
sage: globs['doctest_var']
```

(continues on next page)

(continued from previous page)

```
42
sage: globs.set
{'doctest_var'}
sage: globs.got
{'Integer'}
```

Now we can execute some more doctests to see the dependencies.

```
sage: ex1 = doctests[0].examples[1]
sage: def compiler(ex):
....:     return compile(ex.source, '<doctest sage.doctest.forker[1]>',
....:                    'single', flags, 1)
sage: DTR.compile_and_execute(ex1, compiler, globs)
sage: sorted(list(globs.set))
['R', 'a']
sage: globs.got
{'ZZ'}
sage: ex1.predecessors
[]
```

```
sage: ex2 = doctests[0].examples[2]
sage: def compiler(ex):
....:     return compile(ex.source, '<doctest sage.doctest.forker[2]>',
....:                    'single', flags, 1)
sage: DTR.compile_and_execute(ex2, compiler, globs)
a + 42
sage: list(globs.set)
[]
sage: sorted(list(globs.got))
['a', 'doctest_var']
sage: set(ex2.predecessors) == set([ex0,ex1])
True
```

report_failure(*out*, *test*, *example*, *got*, *globs*)

Called when a doctest fails.

INPUT:

- `out` – a function for printing
- `test` – a `doctest.DocTest` instance
- `example` – a `doctest.Example` instance in `test`
- `got` – string, the result of running `example`
- `globs` – dictionary of globals, used if in debugging mode

OUTPUT: prints a report to `out`

EXAMPLES:

```
sage: from sage.doctest.parsing import SageOutputChecker
sage: from sage.doctest.forker import SageDocTestRunner
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.doctest.control import DocTestDefaults; DD = DocTestDefaults()
sage: import doctest, sys, os
sage: DTR = SageDocTestRunner(SageOutputChecker(), verbose=True, sage_
↪options=DD, optionflags=doctest.NORMALIZE_WHITESPACE|doctest.ELLIPSIS)
```

(continues on next page)

(continued from previous page)

```
sage: filename = sage.doctest.forker.__file__
sage: FDS = FileDocTestSource(filename, DD)
sage: doctests, extras = FDS.create_doctests(globals())
sage: ex = doctests[0].examples[0]
sage: DTR.no_failure_yet = True
sage: DTR.report_failure(sys.stdout.write, doctests[0], ex, 'BAD ANSWER\n', {}
↪)
**********************************************************************
File ".../sage/doctest/forker.py", line 12, in sage.doctest.forker
Failed example:
    doctest_var = 42; doctest_var^2
Expected:
    1764
Got:
    BAD ANSWER
```

If debugging is turned on this function starts an IPython prompt when a test returns an incorrect answer:

```
sage: sage0.quit()
sage: _ = sage0.eval("import doctest, sys, os, multiprocessing, subprocess")
sage: _ = sage0.eval("from sage.doctest.parsing import SageOutputChecker")
sage: _ = sage0.eval("import sage.doctest.forker as sdf")
sage: _ = sage0.eval("from sage.doctest.control import DocTestDefaults")
sage: _ = sage0.eval("DD = DocTestDefaults(debug=True)")
sage: _ = sage0.eval("ex1 = doctest.Example('a = 17', '')")
sage: _ = sage0.eval("ex2 = doctest.Example('2*a', '1')")
sage: _ = sage0.eval("DT = doctest.DocTest([ex1,ex2], globals(), 'doubling',␣
↪None, 0, None)")
sage: _ = sage0.eval("DTR = sdf.SageDocTestRunner(SageOutputChecker(),␣
↪verbose=False, sage_options=DD, optionflags=doctest.NORMALIZE_
↪WHITESPACE|doctest.ELLIPSIS)")
sage: print(sage0.eval("sdf.init_sage(); DTR.run(DT, clear_globs=False)")) #␣
↪indirect doctest
**********************************************************************
Line 1, in doubling
Failed example:
    2*a
Expected:
    1
Got:
    34
**********************************************************************
Previously executed commands:
sage: sage0._expect.expect('sage: ')    # sage0 just mis-identified the output␣
↪as prompt, synchronize
0
sage: sage0.eval("a")
'...17'
sage: sage0.eval("quit")
'Returning to doctests...TestResults(failed=1, attempted=2)'
```

report_overtime(*out*, *test*, *example*, *got*, *check_timer*)

Called when the `warn_long` option flag is set and a doctest runs longer than the specified time.

INPUT:

- `out` – a function for printing

- `test` – a `doctest.DocTest` instance
- `example` – a `doctest.Example` instance in `test`
- `got` – string; the result of running `example`
- `check_timer` – a *`sage.doctest.util.Timer`* (default: `None`) that measures the time spent checking whether or not the output was correct

OUTPUT: prints a report to `out`

EXAMPLES:

```
sage: from sage.doctest.parsing import SageOutputChecker
sage: from sage.doctest.forker import SageDocTestRunner
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.doctest.control import DocTestDefaults; DD = DocTestDefaults()
sage: from sage.doctest.util import Timer
sage: import doctest, sys, os
sage: DTR = SageDocTestRunner(SageOutputChecker(), verbose=True, sage_
↪options=DD, optionflags=doctest.NORMALIZE_WHITESPACE|doctest.ELLIPSIS)
sage: filename = sage.doctest.forker.__file__
sage: FDS = FileDocTestSource(filename, DD)
sage: doctests, extras = FDS.create_doctests(globals())
sage: ex = doctests[0].examples[0]
sage: ex.cputime = 1.23
sage: ex.walltime = 2.50
sage: check = Timer()
sage: check.cputime = 2.34
sage: check.walltime = 3.12
sage: DTR.report_overtime(sys.stdout.write, doctests[0], ex, 'BAD ANSWER\n',␣
↪check_timer=check)
**********************************************************************
File ".../sage/doctest/forker.py", line 12, in sage.doctest.forker
Warning: slow doctest:
    doctest_var = 42; doctest_var^2
Test ran for 1.23s cpu, 2.50s wall
Check ran for 2.34s cpu, 3.12s wall
```

`report_start` (*out*, *test*, *example*)

Called when an example starts.

INPUT:

- `out` – a function for printing
- `test` – a `doctest.DocTest` instance
- `example` – a `doctest.Example` instance in `test`

OUTPUT: prints a report to `out`

EXAMPLES:

```
sage: from sage.doctest.parsing import SageOutputChecker
sage: from sage.doctest.forker import SageDocTestRunner
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.doctest.control import DocTestDefaults; DD = DocTestDefaults()
sage: import doctest, sys, os
sage: DTR = SageDocTestRunner(SageOutputChecker(), verbose=True, sage_
↪options=DD, optionflags=doctest.NORMALIZE_WHITESPACE|doctest.ELLIPSIS)
sage: filename = sage.doctest.forker.__file__
```

(continues on next page)

(continued from previous page)

```
sage: FDS = FileDocTestSource(filename, DD)
sage: doctests, extras = FDS.create_doctests(globals())
sage: ex = doctests[0].examples[0]
sage: DTR.report_start(sys.stdout.write, doctests[0], ex)
Trying (line 12):    doctest_var = 42; doctest_var^2
Expecting:
    1764
```

report_success(*out*, *test*, *example*, *got*, *check_timer*)

Called when an example succeeds.

INPUT:

- `out` – a function for printing
- `test` – a `doctest.DocTest` instance
- `example` – a `doctest.Example` instance in `test`
- `got` – string; the result of running `example`
- `check_timer` – a *sage.doctest.util.Timer* (default: `None`) that measures the time spent checking whether or not the output was correct

OUTPUT: prints a report to `out`; if in debugging mode, starts an IPython prompt at the point of the failure

EXAMPLES:

```
sage: from sage.doctest.parsing import SageOutputChecker
sage: from sage.doctest.forker import SageDocTestRunner
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.doctest.control import DocTestDefaults; DD = DocTestDefaults()
sage: from sage.doctest.util import Timer
sage: import doctest, sys, os
sage: DTR = SageDocTestRunner(SageOutputChecker(), verbose=True, sage_
↪options=DD, optionflags=doctest.NORMALIZE_WHITESPACE|doctest.ELLIPSIS)
sage: filename = sage.doctest.forker.__file__
sage: FDS = FileDocTestSource(filename, DD)
sage: doctests, extras = FDS.create_doctests(globals())
sage: ex = doctests[0].examples[0]
sage: ex.cputime = 1.01
sage: ex.walltime = 1.12
sage: check = Timer()
sage: check.cputime = 2.14
sage: check.walltime = 2.71
sage: DTR.report_success(sys.stdout.write, doctests[0], ex, '1764',
....:                    check_timer=check)
ok [3.83s wall]
```

report_unexpected_exception(*out*, *test*, *example*, *exc_info*)

Called when a doctest raises an exception that's not matched by the expected output.

If debugging has been turned on, starts an interactive debugger.

INPUT:

- `out` – a function for printing
- `test` – a `doctest.DocTest` instance
- `example` – a `doctest.Example` instance in `test`

- `exc_info` – the result of `sys.exc_info()`

OUTPUT: prints a report to `out`

- if in debugging mode, starts PDB with the given traceback

EXAMPLES:

```
sage: from sage.interfaces.sage0 import sage0
sage: sage0.quit()
sage: _ = sage0.eval("import doctest, sys, os, multiprocessing, subprocess")
sage: _ = sage0.eval("from sage.doctest.parsing import SageOutputChecker")
sage: _ = sage0.eval("import sage.doctest.forker as sdf")
sage: _ = sage0.eval("from sage.doctest.control import DocTestDefaults")
sage: _ = sage0.eval("DD = DocTestDefaults(debug=True)")
sage: _ = sage0.eval("ex = doctest.Example('E = EllipticCurve([0,0]); E', 'A␣
↪singular Elliptic Curve')")
sage: _ = sage0.eval("DT = doctest.DocTest([ex], globals(), 'singular_curve',␣
↪None, 0, None)")
sage: _ = sage0.eval("DTR = sdf.SageDocTestRunner(SageOutputChecker(),␣
↪verbose=False, sage_options=DD, optionflags=doctest.NORMALIZE_
↪WHITESPACE|doctest.ELLIPSIS)")
sage: old_prompt = sage0._prompt
sage: sage0._prompt = r"\(Pdb\) "
sage: sage0.eval("DTR.run(DT, clear_globs=False)") # indirect doctest
'... ArithmeticError(self._equation_string() + " defines a singular curve")'
sage: sage0.eval("l")
'...if self.discriminant() == 0:...raise ArithmeticError...'
sage: sage0.eval("u")
'...-> super().__init__(R, data, category=category)'
sage: sage0.eval("u")
'...EllipticCurve_field.__init__(self, K, ainvs)'
sage: sage0.eval("p ainvs")
'(0, 0, 0, 0, 0)'
sage: sage0._prompt = old_prompt
sage: sage0.eval("quit")
'TestResults(failed=1, attempted=1)'
```

run (*test*, *compileflags=0*, *out=None*, *clear_globs=True*)

Run the examples in a given doctest.

This function replaces `doctest.DocTestRunner.run` since it needs to handle spoofing. It also leaves the display hook in place.

INPUT:

- `test` – an instance of `doctest.DocTest`
- `compileflags` – integer (default: 0) the set of compiler flags used to execute examples (passed in to the `compile()`)
- `out` – a function for writing the output (defaults to `sys.stdout.write()`)
- `clear_globs` – boolean (default: `True`); whether to clear the namespace after running this doctest

OUTPUT:

- `f` – integer, the number of examples that failed
- `t` – the number of examples tried

EXAMPLES:

```
sage: from sage.doctest.parsing import SageOutputChecker
sage: from sage.doctest.forker import SageDocTestRunner
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.doctest.control import DocTestDefaults; DD = DocTestDefaults()
sage: import doctest, sys, os
sage: DTR = SageDocTestRunner(SageOutputChecker(), verbose=False, sage_
↪options=DD,
....:                         optionflags=doctest.NORMALIZE_
↪WHITESPACE|doctest.ELLIPSIS)
sage: filename = sage.doctest.forker.__file__
sage: FDS = FileDocTestSource(filename, DD)
sage: doctests, extras = FDS.create_doctests(globals())
sage: DTR.run(doctests[0], clear_globs=False)
TestResults(failed=0, attempted=4)
```

summarize(*verbose=None*)

Print results of testing to `self.msgfile` and return number of failures and tests run.

INPUT:

- `verbose` – whether to print lots of stuff

OUTPUT:

- returns `(f, t)`, a `doctest.TestResults` instance giving the number of failures and the total number of tests run.

EXAMPLES:

```
sage: from sage.doctest.parsing import SageOutputChecker
sage: from sage.doctest.forker import SageDocTestRunner
sage: from sage.doctest.control import DocTestDefaults; DD = DocTestDefaults()
sage: import doctest, sys, os
sage: DTR = SageDocTestRunner(SageOutputChecker(), verbose=False, sage_
↪options=DD, optionflags=doctest.NORMALIZE_WHITESPACE|doctest.ELLIPSIS)
sage: DTR._name2ft['sage.doctest.forker'] = (1,120)
sage: results = DTR.summarize()
**********************************************************************
1 item had failures:
    1 of 120 in sage.doctest.forker
sage: results
TestResults(failed=1, attempted=120)
```

update_digests(*example*)

Update global and doctest digests.

Sage's doctest runner tracks the state of doctests so that their dependencies are known. For example, in the following two lines

```
sage: R.<x> = ZZ[]
sage: f = x^2 + 1
```

it records that the second line depends on the first since the first INSERTS `x` into the global namespace and the second line RETRIEVES `x` from the global namespace.

This function updates the hashes that record these dependencies.

INPUT:

- `example` – a `doctest.Example` instance

EXAMPLES:

```
sage: from sage.doctest.parsing import SageOutputChecker
sage: from sage.doctest.forker import SageDocTestRunner
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.doctest.control import DocTestDefaults; DD = DocTestDefaults()
sage: import doctest, sys, os, hashlib
sage: DTR = SageDocTestRunner(SageOutputChecker(), verbose=False, sage_
↪options=DD, optionflags=doctest.NORMALIZE_WHITESPACE|doctest.ELLIPSIS)
sage: filename = sage.doctest.forker.__file__
sage: FDS = FileDocTestSource(filename, DD)
sage: doctests, extras = FDS.create_doctests(globals())
sage: DTR.running_global_digest.hexdigest()
'd41d8cd98f00b204e9800998ecf8427e'
sage: DTR.running_doctest_digest = hashlib.md5()
sage: ex = doctests[0].examples[0]; ex.predecessors = None
sage: DTR.update_digests(ex)
sage: DTR.running_global_digest.hexdigest()
'3cb44104292c3a3ab4da3112ce5dc35c'
```

update_results(*D*)

When returning results we pick out the results of interest since many attributes are not pickleable.

INPUT:

- `D` – dictionary to update with cputime and walltime

OUTPUT: the number of failures (or `False` if there is no failure attribute)

EXAMPLES:

```
sage: from sage.doctest.parsing import SageOutputChecker
sage: from sage.doctest.forker import SageDocTestRunner
sage: from sage.doctest.sources import FileDocTestSource, DictAsObject
sage: from sage.doctest.control import DocTestDefaults; DD = DocTestDefaults()
sage: import doctest, sys, os
sage: DTR = SageDocTestRunner(SageOutputChecker(), verbose=False, sage_
↪options=DD, optionflags=doctest.NORMALIZE_WHITESPACE|doctest.ELLIPSIS)
sage: filename = sage.doctest.forker.__file__
sage: FDS = FileDocTestSource(filename, DD)
sage: doctests, extras = FDS.create_doctests(globals())
sage: from sage.doctest.util import Timer
sage: T = Timer().start()
sage: DTR.run(doctests[0])
TestResults(failed=0, attempted=4)
sage: T.stop().annotate(DTR)
sage: D = DictAsObject({'cputime': [], 'walltime': [], 'err': None})
sage: DTR.update_results(D)
0
sage: sorted(list(D.items()))
[('cputime', [...]), ('err', None), ('failures', 0), ('tests', 4),
 ('walltime', [...]), ('walltime_skips', 0)]
```

class sage.doctest.forker.**SageSpoofInOut**(*outfile=None*, *infile=None*)

Bases: `SageObject`

We replace the standard `doctest._SpoofOut` for three reasons:

- we need to divert the output of C programs that don't print through sys.stdout,
- we want the ability to recover partial output from doctest processes that segfault.

- we also redirect stdin (usually from /dev/null) during doctests.

This class defines streams `self.real_stdin`, `self.real_stdout` and `self.real_stderr` which refer to the original streams.

INPUT:

- `outfile` – (default: `tempfile.TemporaryFile()`) a seekable open file object to which stdout and stderr should be redirected
- `infile` – (default: `open(os.devnull)`) an open file object from which stdin should be redirected

EXAMPLES:

```
sage: import subprocess, tempfile
sage: from sage.doctest.forker import SageSpoofInOut
sage: O = tempfile.TemporaryFile()
sage: S = SageSpoofInOut(O)
sage: try:
....:     S.start_spoofing()
....:     print("hello world")
....: finally:
....:     S.stop_spoofing()
....:
sage: S.getvalue()
'hello world\n'
sage: _ = O.seek(0)
sage: S = SageSpoofInOut(outfile=sys.stdout, infile=O)
sage: try:
....:     S.start_spoofing()
....:     _ = subprocess.check_call("cat")
....: finally:
....:     S.stop_spoofing()
....:
hello world
sage: O.close()
```

getvalue()

Get the value that has been printed to `outfile` since the last time this function was called.

EXAMPLES:

```
sage: from sage.doctest.forker import SageSpoofInOut
sage: S = SageSpoofInOut()
sage: try:
....:     S.start_spoofing()
....:     print("step 1")
....: finally:
....:     S.stop_spoofing()
....:
sage: S.getvalue()
'step 1\n'
sage: try:
....:     S.start_spoofing()
....:     print("step 2")
....: finally:
....:     S.stop_spoofing()
....:
sage: S.getvalue()
'step 2\n'
```

start_spoofing()

Set stdin to read from `self.infile` and stdout to print to `self.outfile`.

EXAMPLES:

```
sage: import os, tempfile
sage: from sage.doctest.forker import SageSpoofInOut
sage: O = tempfile.TemporaryFile()
sage: S = SageSpoofInOut(O)
sage: try:
....:     S.start_spoofing()
....:     print("this is not printed")
....: finally:
....:     S.stop_spoofing()
....:
sage: S.getvalue()
'this is not printed\n'
sage: _ = O.seek(0)
sage: S = SageSpoofInOut(infile=O)
sage: try:
....:     S.start_spoofing()
....:     v = sys.stdin.read()
....: finally:
....:     S.stop_spoofing()
....:
sage: v
'this is not printed\n'
```

We also catch non-Python output:

```
sage: try:
....:     S.start_spoofing()
....:     retval = os.system('''echo "Hello there"\nif [ $? -eq 0 ]; then\
↪necho "good"\nfi''')
....: finally:
....:     S.stop_spoofing()
....:
sage: S.getvalue()
'Hello there\ngood\n'
sage: O.close()
```

stop_spoofing()

Reset stdin and stdout to their original values.

EXAMPLES:

```
sage: from sage.doctest.forker import SageSpoofInOut
sage: S = SageSpoofInOut()
sage: try:
....:     S.start_spoofing()
....:     print("this is not printed")
....: finally:
....:     S.stop_spoofing()
....:
sage: print("this is now printed")
this is now printed
```

class sage.doctest.forker.**TestResults**(*failed*, *attempted*)

Bases: `tuple`

attempted

Alias for field number 1

failed

Alias for field number 0

sage.doctest.forker.**dummy_handler**(*sig*, *frame*)

Dummy signal handler for SIGCHLD (just to ensure the signal isn't ignored).

sage.doctest.forker.**init_sage**(*controller=None*)

Import the Sage library.

This function is called once at the beginning of a doctest run (rather than once for each file). It imports the Sage library, sets DOCTEST_MODE to True, and invalidates any interfaces.

EXAMPLES:

```
sage: from sage.doctest.forker import init_sage
sage: sage.doctest.DOCTEST_MODE = False
sage: init_sage()
sage: sage.doctest.DOCTEST_MODE
True
```

Check that pexpect interfaces are invalidated, but still work:

```
sage: gap.eval("my_test_var := 42;")
'42'
sage: gap.eval("my_test_var;")
'42'
sage: init_sage()
sage: gap('Group((1,2,3)(4,5), (3,4))')
Group( [ (1,2,3)(4,5), (3,4) ] )
sage: gap.eval("my_test_var;")
Traceback (most recent call last):
...
RuntimeError: Gap produced error output...
```

Check that SymPy equation pretty printer is limited in doctest mode to default width (80 chars):

```
sage: # needs sympy
sage: from sympy import sympify
sage: from sympy.printing.pretty.pretty import PrettyPrinter
sage: s = sympify('+x^'.join(str(i) for i in range(30)))
sage: print(PrettyPrinter(settings={'wrap_line': True}).doprint(s))
 29    28    27    26    25    24    23    22    21    20    19    18    17...
x   + x   + x   + x   + x   + x   + x   + x   + x   + x   + x   + x   + x...

... 16    15    14    13    12    11    10    9    8    7    6    5    4    3...
...x   + x   + x   + x   + x   + x   + x   + x  + x  + x  + x  + x  + x  + x...

...
```

The displayhook sorts dictionary keys to simplify doctesting of dictionary output:

```
sage: {'a':23, 'b':34, 'au':56, 'bbf':234, 'aaa':234}
{'a': 23, 'aaa': 234, 'au': 56, 'b': 34, 'bbf': 234}
```

sage.doctest.forker.**showwarning_with_traceback**(*message*, *category*, *filename*, *lineno*, *file=None*, *line=None*)

Displays a warning message with a traceback.

INPUT: see `warnings.showwarning()`.

OUTPUT: none

EXAMPLES:

```
sage: from sage.doctest.forker import showwarning_with_traceback
sage: showwarning_with_traceback("bad stuff", UserWarning, "myfile.py", 0)
doctest:warning...
  File "<doctest sage.doctest.forker.showwarning_with_traceback[1]>", line 1, in
↪<module>
    showwarning_with_traceback("bad stuff", UserWarning, "myfile.py", Integer(0))
:
UserWarning: bad stuff
```

CHAPTER

FOUR

PARSING DOCSTRINGS

This module contains functions and classes that parse docstrings.

AUTHORS:

- David Roe (2012-03-27) – initial version, based on Robert Bradshaw's code.
- Jeroen Demeyer(2014-08-28) – much improved handling of tolerances using interval arithmetic (Issue #16889).

class sage.doctest.parsing.**OriginalSource**(*example*)

Bases: object

Context swapping out the pre-parsed source with the original for better reporting.

EXAMPLES:

```
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.doctest.control import DocTestDefaults
sage: filename = sage.doctest.forker.__file__
sage: FDS = FileDocTestSource(filename, DocTestDefaults())
sage: doctests, extras = FDS.create_doctests(globals())
sage: ex = doctests[0].examples[0]
sage: ex.sage_source
'doctest_var = 42; doctest_var^2\n'
sage: ex.source
'doctest_var = Integer(42); doctest_var**Integer(2)\n'
sage: from sage.doctest.parsing import OriginalSource
sage: with OriginalSource(ex):
....:     ex.source
'doctest_var = 42; doctest_var^2\n'
```

class sage.doctest.parsing.**SageDocTestParser**(*optional_tags=()*, *long=False*, ***, *probed_tags=()*, *file_optional_tags=()*)

Bases: DocTestParser

A version of the standard doctest parser which handles Sage's custom options and tolerances in floating point arithmetic.

file_optional_tags: set[str]

long: bool

optional_only: bool

optional_tags: bool | set[str]

optionals: dict[str, int]

parse(*string*, **args*)

A Sage specialization of `doctest.DocTestParser`.

INPUT:

- `string` – the string to parse
- `name` – (optional) string giving the name identifying string, to be used in error messages

OUTPUT:

- A list consisting of strings and `doctest.Example` instances. There will be at least one string between successive examples (exactly one unless long or optional tests are removed), and it will begin and end with a string.

EXAMPLES:

```
sage: from sage.doctest.parsing import SageDocTestParser
sage: DTP = SageDocTestParser(('sage','magma','guava'))
sage: example = 'Explanatory text::\n\n    sage: E = magma("EllipticCurve([1,␣
↪1, 1, -10, -10])") # optional: magma\n\nLater text'
sage: parsed = DTP.parse(example)
sage: parsed[0]
'Explanatory text::\n\n'
sage: parsed[1].sage_source
'E = magma("EllipticCurve([1, 1, 1, -10, -10])") # optional: magma\n'
sage: parsed[2]
'\nLater text'
```

If the doctest parser is not created to accept a given optional argument, the corresponding examples will just be removed:

```
sage: DTP2 = SageDocTestParser(('sage',))
sage: parsed2 = DTP2.parse(example)
sage: parsed2
['Explanatory text::\n\n', '\nLater text']
```

You can mark doctests as having a particular tolerance:

```
sage: example2 = 'sage: gamma(1.6) # tol 2.0e-11\n0.893515349287690'
sage: ex = DTP.parse(example2)[1]
sage: ex.sage_source
'gamma(1.6) # tol 2.0e-11\n'
sage: ex.want
'0.893515349287690\n'
sage: type(ex.want)
<class 'sage.doctest.marked_output.MarkedOutput'>
sage: ex.want.tol
2.000000000000000000...?e-11
```

You can use continuation lines:

```
sage: s = "sage: for i in range(4):\n....:     print(i)\n....:\n"
sage: ex = DTP2.parse(s)[1]
sage: ex.source
'for i in range(Integer(4)):\n    print(i)\n'
```

Sage currently accepts backslashes as indicating that the end of the current line should be joined to the next line. This feature allows for breaking large integers over multiple lines but is not standard for Python doctesting. It's not guaranteed to persist:

```
sage: n = 1234\
....:     5678
sage: print(n)
12345678
sage: type(n)
<class 'sage.rings.integer.Integer'>
```

It also works without the line continuation:

```
sage: m = 8765\
4321
sage: print(m)
87654321
```

Optional tags at the start of an example block persist to the end of the block (delimited by a blank line):

```
sage: # long time, needs sage.rings.number_field
sage: QQbar(I)^10000
1
sage: QQbar(I)^10000  # not tested
I

sage: # needs sage.rings.finite_rings
sage: GF(7)
Finite Field of size 7
sage: GF(10)
Traceback (most recent call last):
...
ValueError: the order of a finite field must be a prime power
```

Test that Issue #26575 is resolved:

```
sage: example3 = 'sage: Zp(5,4,print_mode="digits")(5)\n...00010'
sage: parsed3 = DTP.parse(example3)
sage: dte = parsed3[1]
sage: dte.sage_source
'Zp(5,4,print_mode="digits")(5)\n'
sage: dte.want
'...00010\n'
```

Style warnings:

```
sage: def parse(test_string):
....:     return [x if isinstance(x, str)
....:               else (getattr(x, 'warnings', None), x.sage_source, x.
↪source)
....:             for x in DTP.parse(test_string)]

sage: parse('sage: 1 # optional guava mango\nsage: 2 # optional guava\nsage:␣
↪3 # optional guava\nsage: 4 # optional guava\nsage: 5 # optional guava\n\
↪nsage: 11 # optional guava')
['',
 (["Consider using a block-scoped tag by inserting the line 'sage: # optional␣
↪- guava' just before this line to avoid repeating the tag 5 times"],
  '1 # optional guava mango\n',
  'None  # virtual doctest'),
 '',
```

(continues on next page)

(continued from previous page)

```
 (None, '2 # optional guava\n', 'Integer(2) # optional guava\n'),
 '',
 (None, '3 # optional guava\n', 'Integer(3) # optional guava\n'),
 '',
 (None, '4 # optional guava\n', 'Integer(4) # optional guava\n'),
 '',
 (None, '5 # optional guava\n', 'Integer(5) # optional guava\n'),
 '\n',
 (None, '11 # optional guava\n', 'Integer(11) # optional guava\n'),
 '']

sage: parse('sage: 1 # optional guava\nsage: 2 # optional guava mango\nsage:␣
↪3 # optional guava\nsage: 4 # optional guava\nsage: 5 # optional guava\n')
['',
 (["Consider using a block-scoped tag by inserting the line 'sage: # optional␣
↪- guava' just before this line to avoid repeating the tag 5 times"],
  '1 # optional guava\n',
  'Integer(1) # optional guava\n'),
 '',
 '',
 (None, '3 # optional guava\n', 'Integer(3) # optional guava\n'),
 '',
 (None, '4 # optional guava\n', 'Integer(4) # optional guava\n'),
 '',
 (None, '5 # optional guava\n', 'Integer(5) # optional guava\n'),
 '']

sage: parse('sage: # optional mango\nsage: 1 # optional guava\nsage: 2 #␣
↪optional guava mango\nsage: 3 # optional guava\nsage: 4 # optional guava\n␣
↪sage: 5 # optional guava\n')  # optional - guava mango
['',
 (["Consider updating this block-scoped tag to 'sage: # optional - guava mango
↪' to avoid repeating the tag 5 times"],
  '# optional mango\n',
  'None  # virtual doctest'),
 '',
 '',
 '',
 '',
 '',
 '']

sage: parse('::\n\n    sage: 1 # optional guava\n    sage: 2 # optional guava␣
↪mango\n    sage: 3 # optional guava\n\n::\n\n    sage: 4 # optional guava\n␣
↪    sage: 5 # optional guava\n')
['::\n\n',
(None, '1 # optional guava\n', 'Integer(1) # optional guava\n'),
'',
'',
(None, '3 # optional guava\n', 'Integer(3) # optional guava\n'),
'\n::\n\n',
(None, '4 # optional guava\n', 'Integer(4) # optional guava\n'),
'',
(None, '5 # optional guava\n', 'Integer(5) # optional guava\n'),
'']
```

probed_tags: bool | set[str]

class sage.doctest.parsing.**SageOutputChecker**

Bases: OutputChecker

A modification of the doctest OutputChecker that can check relative and absolute tolerance of answers.

EXAMPLES:

```
sage: from sage.doctest.parsing import SageOutputChecker, MarkedOutput,␣
↪SageDocTestParser
sage: import doctest
sage: optflag = doctest.NORMALIZE_WHITESPACE|doctest.ELLIPSIS
sage: DTP = SageDocTestParser(('sage','magma','guava'))
sage: OC = SageOutputChecker()
sage: example2 = 'sage: gamma(1.6) # tol 2.0e-11\n0.893515349287690'
sage: ex = DTP.parse(example2)[1]
sage: ex.sage_source
'gamma(1.6) # tol 2.0e-11\n'
sage: ex.want
'0.893515349287690\n'
sage: type(ex.want)
<class 'sage.doctest.marked_output.MarkedOutput'>
sage: ex.want.tol
2.000000000000000000...?e-11
sage: OC.check_output(ex.want, '0.893515349287690', optflag)
True
sage: OC.check_output(ex.want, '0.8935153492877', optflag)
True
sage: OC.check_output(ex.want, '0', optflag)
False
sage: OC.check_output(ex.want, 'x + 0.8935153492877', optflag)
False
```

check_output(*want*, *got*, *optionflags*)

Check to see if the output matches the desired output.

If `want` is a `MarkedOutput` instance, takes into account the desired tolerance.

INPUT:

- `want` – string or `MarkedOutput`
- `got` – string
- `optionflags` – integer; passed down to `doctest.OutputChecker`

OUTPUT: boolean; whether `got` matches `want` up to the specified tolerance

EXAMPLES:

```
sage: from sage.doctest.parsing import MarkedOutput, SageOutputChecker
sage: import doctest
sage: optflag = doctest.NORMALIZE_WHITESPACE|doctest.ELLIPSIS
sage: rndstr = MarkedOutput("I'm wrong!").update(random=True)
sage: tentol = MarkedOutput("10.0").update(tol=.1)
sage: tenabs = MarkedOutput("10.0").update(abs_tol=.1)
sage: tenrel = MarkedOutput("10.0").update(rel_tol=.1)
sage: zerotol = MarkedOutput("0.0").update(tol=.1)
sage: zeroabs = MarkedOutput("0.0").update(abs_tol=.1)
sage: zerorel = MarkedOutput("0.0").update(rel_tol=.1)
sage: zero = "0.0"
sage: nf = "9.5"
```

(continues on next page)

(continued from previous page)

```
sage: ten = "10.05"
sage: eps = "-0.05"
sage: OC = SageOutputChecker()
```

```
sage: OC.check_output(rndstr,nf,optflag)
True

sage: OC.check_output(tentol,nf,optflag)
True
sage: OC.check_output(tentol,ten,optflag)
True
sage: OC.check_output(tentol,zero,optflag)
False

sage: OC.check_output(tenabs,nf,optflag)
False
sage: OC.check_output(tenabs,ten,optflag)
True
sage: OC.check_output(tenabs,zero,optflag)
False

sage: OC.check_output(tenrel,nf,optflag)
True
sage: OC.check_output(tenrel,ten,optflag)
True
sage: OC.check_output(tenrel,zero,optflag)
False

sage: OC.check_output(zerotol,zero,optflag)
True
sage: OC.check_output(zerotol,eps,optflag)
True
sage: OC.check_output(zerotol,ten,optflag)
False

sage: OC.check_output(zeroabs,zero,optflag)
True
sage: OC.check_output(zeroabs,eps,optflag)
True
sage: OC.check_output(zeroabs,ten,optflag)
False

sage: OC.check_output(zerorel,zero,optflag)
True
sage: OC.check_output(zerorel,eps,optflag)
False
sage: OC.check_output(zerorel,ten,optflag)
False
```

More explicit tolerance checks:

```
sage: _ = x  # rel tol 1e10                                                    #␣
↪needs sage.symbolic
sage: raise RuntimeError   # rel tol 1e10
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
RuntimeError
sage: 1  # abs tol 2
-0.5
sage: print("0.9999")     # rel tol 1e-4
1.0
sage: print("1.00001")    # abs tol 1e-5
1.0
sage: 0  # rel tol 1
1
```

Abs tol checks over the complex domain:

```
sage: [1, -1.3, -1.5 + 0.1*I, 0.5 - 0.1*I, -1.5*I]  # abs tol 1.0
[1, -1, -1, 1, -I]
```

Spaces before numbers or between the sign and number are ignored:

```
sage: print("[ - 1, 2]")  # abs tol 1e-10
[-1,2]
```

Tolerance on Python 3 for string results with unicode prefix:

```
sage: a = 'Cyrano'; a
'Cyrano'
sage: b = ['Fermat', 'Euler']; b
['Fermat',  'Euler']
sage: c = 'you'; c
'you'
```

This illustrates that Issue #33588 is fixed:

```
sage: from sage.doctest.parsing import SageOutputChecker, SageDocTestParser
sage: import doctest
sage: optflag = doctest.NORMALIZE_WHITESPACE|doctest.ELLIPSIS
sage: DTP = SageDocTestParser(('sage','magma','guava'))
sage: OC = SageOutputChecker()
sage: example = "sage: 1.3090169943749475 # tol 1e-8\n1.3090169943749475"
sage: ex = DTP.parse(example)[1]
sage: OC.check_output(ex.want, '1.3090169943749475', optflag)
True
sage: OC.check_output(ex.want, 'ANYTHING1.3090169943749475', optflag)
False
sage: OC.check_output(ex.want, 'Long-step dual simplex will be used\n1.
↪3090169943749475', optflag)
True
```

do_fixup(*want*, *got*)

Perform few changes to the strings `want` and `got`.

For example, remove warnings to be ignored.

INPUT:

- `want` – string or `MarkedOutput`
- `got` – string

OUTPUT: a tuple:

- boolean, `True` when some fixup were performed and `False` otherwise
- string, edited wanted string
- string, edited got string

Note

Currently, the code only possibly changes the string `got` while keeping `want` invariant. We keep open the possibility of adding a regular expression which would also change the `want` string. This is why `want` is an input and an output of the method even if currently kept invariant.

EXAMPLES:

```
sage: from sage.doctest.parsing import SageOutputChecker
sage: OC = SageOutputChecker()
sage: OC.do_fixup('1.3090169943749475','1.3090169943749475')
(False, '1.3090169943749475', '1.3090169943749475')
sage: OC.do_fixup('1.3090169943749475','ANYTHING1.3090169943749475')
(False, '1.3090169943749475', 'ANYTHING1.3090169943749475')
sage: OC.do_fixup('1.3090169943749475','Long-step dual simplex will be used\
↪n1.3090169943749475')
(True, '1.3090169943749475', '\n1.3090169943749475')
```

When `want` is an instance of class `MarkedOutput`:

```
sage: from sage.doctest.parsing import SageOutputChecker, SageDocTestParser
sage: import doctest
sage: optflag = doctest.NORMALIZE_WHITESPACE|doctest.ELLIPSIS
sage: DTP = SageDocTestParser(('sage','magma','guava'))
sage: OC = SageOutputChecker()
sage: example = "sage: 1.3090169943749475\n1.3090169943749475"
sage: ex = DTP.parse(example)[1]
sage: ex.want
'1.3090169943749475\n'
sage: OC.do_fixup(ex.want,'1.3090169943749475')
(False, '1.3090169943749475\n', '1.3090169943749475')
sage: OC.do_fixup(ex.want,'ANYTHING1.3090169943749475')
(False, '1.3090169943749475\n', 'ANYTHING1.3090169943749475')
sage: OC.do_fixup(ex.want,'Long-step dual simplex will be used\n1.
↪3090169943749475')
(True, '1.3090169943749475\n', '\n1.3090169943749475')
```

human_readable_escape_sequences(*string*)

Make ANSI escape sequences human readable.

EXAMPLES:

```
sage: print('This is \x1b[1mbold\x1b[0m text')
This is <CSI-1m>bold<CSI-0m> text
```

output_difference(*example*, *got*, *optionflags*)

Report on the differences between the desired result and what was actually obtained.

If `want` is a `MarkedOutput` instance, takes into account the desired tolerance.

INPUT:

- `example` – a `doctest.Example` instance
- `got` – string
- `optionflags` – integer; passed down to `doctest.OutputChecker`

OUTPUT: string, describing how `got` fails to match `example.want`

EXAMPLES:

```
sage: from sage.doctest.parsing import MarkedOutput, SageOutputChecker
sage: import doctest
sage: optflag = doctest.NORMALIZE_WHITESPACE|doctest.ELLIPSIS
sage: tentol = doctest.Example('',MarkedOutput("10.0\n").update(tol=.1))
sage: tenabs = doctest.Example('',MarkedOutput("10.0\n").update(abs_tol=.1))
sage: tenrel = doctest.Example('',MarkedOutput("10.0\n").update(rel_tol=.1))
sage: zerotol = doctest.Example('',MarkedOutput("0.0\n").update(tol=.1))
sage: zeroabs = doctest.Example('',MarkedOutput("0.0\n").update(abs_tol=.1))
sage: zerorel = doctest.Example('',MarkedOutput("0.0\n").update(rel_tol=.1))
sage: tlist = doctest.Example('',MarkedOutput("[10.0, 10.0, 10.0, 10.0, 10.0,␣
↪10.0]\n").update(abs_tol=0.987))
sage: zero = "0.0"
sage: nf = "9.5"
sage: ten = "10.05"
sage: eps = "-0.05"
sage: L = "[9.9, 8.7, 10.3, 11.2, 10.8, 10.0]"
sage: OC = SageOutputChecker()
```

```
sage: print(OC.output_difference(tenabs,nf,optflag))
Expected:
    10.0
Got:
    9.5
Tolerance exceeded:
    10.0 vs 9.5, tolerance 5e-1 > 1e-1

sage: print(OC.output_difference(tentol,zero,optflag))
Expected:
    10.0
Got:
    0.0
Tolerance exceeded:
    10.0 vs 0.0, tolerance 1e0 > 1e-1

sage: print(OC.output_difference(tentol,eps,optflag))
Expected:
    10.0
Got:
    -0.05
Tolerance exceeded:
    10.0 vs -0.05, tolerance 2e0 > 1e-1

sage: print(OC.output_difference(tlist,L,optflag))
Expected:
    [10.0, 10.0, 10.0, 10.0, 10.0, 10.0]
Got:
    [9.9, 8.7, 10.3, 11.2, 10.8, 10.0]
Tolerance exceeded in 2 of 6:
    10.0 vs 8.7, tolerance 2e0 > 9.87e-1
```

(continues on next page)

(continued from previous page)

```
    10.0 vs 11.2, tolerance 2e0 > 9.87e-1
```

sage.doctest.parsing.**get_source**(*example*)

Return the source with the leading 'sage: ' stripped off.

EXAMPLES:

```
sage: from sage.doctest.parsing import get_source
sage: from sage.doctest.sources import DictAsObject
sage: example = DictAsObject({})
sage: example.sage_source = "2 + 2"
sage: example.source = "sage: 2 + 2"
sage: get_source(example)
'2 + 2'
sage: example = DictAsObject({})
sage: example.source = "3 + 3"
sage: get_source(example)
'3 + 3'
```

sage.doctest.parsing.**parse_file_optional_tags**(*lines*)

Scan the first few lines for file-level doctest directives.

INPUT:

- `lines` – iterable of pairs `(lineno, line)`

OUTPUT: dictionary whose keys are strings (tags); see *parse_optional_tags()*

EXAMPLES:

```
sage: from sage.doctest.parsing import parse_file_optional_tags
sage: filename = tmp_filename(ext='.pyx')
sage: with open(filename, "r") as f:
....:     parse_file_optional_tags(enumerate(f))
{}
sage: with open(filename, "w") as f:
....:     _ = f.write("# nodoctest")
sage: with open(filename, "r") as f:
....:     parse_file_optional_tags(enumerate(f))
{'not tested': None}
sage: with open(filename, "w") as f:
....:     _ = f.write("# sage.doctest: "    # broken in two source lines to avoid␣
↪the pattern
....:                 "optional - xyz")     # of relint (multiline_doctest_
↪comment)
sage: with open(filename, "r") as f:
....:     parse_file_optional_tags(enumerate(f))
{'xyz': None}
```

sage.doctest.parsing.**parse_optional_tags**(*string: str*) → dict[str, str | None]

sage.doctest.parsing.**parse_optional_tags**(*string: str*, *, *return_string_sans_tags: Literal[True]*) → tuple[dict[str, str | None], str, bool]

Return a dictionary whose keys are optional tags from the following set that occur in a comment on the first line of the input string.

- `'long time'`
- `'not implemented'`

- 'not tested'
- 'known bug' (possible values are None, linux and macos)
- 'py2'
- 'optional -- FEATURE...' or 'needs FEATURE...' – the dictionary will just have the key 'FEATURE'

The values, if non-None, are strings with optional explanations for a tag, which may appear in parentheses after the tag in string.

INPUT:

- string – string
- return_string_sans_tags – boolean (default: False); whether to additionally return string with the optional tags removed but other comments kept and a boolean is_persistent

EXAMPLES:

```
sage: from sage.doctest.parsing import parse_optional_tags
sage: parse_optional_tags("sage: magma('2 + 2')# optional: magma")
{'magma': None}
sage: parse_optional_tags("sage: #optional -- mypkg")
{'mypkg': None}
sage: parse_optional_tags("sage: print(1)  # parentheses are optional here")
{}
sage: parse_optional_tags("sage: print(1)  # optional")
{}
sage: sorted(list(parse_optional_tags("sage: #optional -- foo bar, baz")))
['bar', 'foo']
sage: parse_optional_tags("sage: #optional -- foo.bar, baz")
{'foo.bar': None}
sage: parse_optional_tags("sage: #needs foo.bar, baz")
{'foo.bar': None}
sage: sorted(list(parse_optional_tags("    sage: factor(10^(10^10) + 1) # LoNg␣
↪TiME, NoT TeSTED; OptioNAL -- P4cka9e")))
['long time', 'not tested', 'p4cka9e']
sage: parse_optional_tags("    sage: raise RuntimeError # known bug")
{'bug': None}
sage: sorted(list(parse_optional_tags("    sage: determine_meaning_of_life() #␣
↪long time, not implemented")))
['long time', 'not implemented']
```

We don't parse inside strings:

```
sage: parse_optional_tags("    sage: print('  # long time')")
{}
sage: parse_optional_tags("    sage: print('  # long time')  # not tested")
{'not tested': None}
```

UTF-8 works:

```
sage: parse_optional_tags("'ěščřžýáíédĎ'")
{}
```

Tags with parenthesized explanations:

```
sage: parse_optional_tags("    sage: 1 + 1  # long time (1 year, 2 months??),␣
↪optional - bliss (because)")
{'bliss': 'because', 'long time': '1 year, 2 months??'}
```

With `return_string_sans_tags=True`:

```
sage: parse_optional_tags("sage: print(1)  # very important 1  # optional - foo",
....:                     return_string_sans_tags=True)
({'foo': None}, 'sage: print(1)  # very important 1  ', False)
sage: parse_optional_tags("sage: print(    # very important too  # optional - foo\
↪n....:     2)",
....:                     return_string_sans_tags=True)
({'foo': None}, 'sage: print(    # very important too  \n....:     2)', False)
sage: parse_optional_tags("sage: #this is persistent #needs scipy",
....:                     return_string_sans_tags=True)
({'scipy': None}, 'sage: #this is persistent ', True)
sage: parse_optional_tags("sage: #this is not #needs scipy\n....: import scipy",
....:                     return_string_sans_tags=True)
({'scipy': None}, 'sage: #this is not \n....: import scipy', False)
```

sage.doctest.parsing.**parse_tolerance**(*source*, *want*)

Return a version of `want` marked up with the tolerance tags specified in `source`.

INPUT:

- `source` – string, the source of a doctest
- `want` – string, the desired output of the doctest

OUTPUT: `want` if there are no tolerance tags specified; a `MarkedOutput` version otherwise

EXAMPLES:

```
sage: from sage.doctest.parsing import parse_tolerance
sage: marked = parse_tolerance("sage: s.update(abs_tol = .0000001)", "")
sage: type(marked)
<class 'str'>
sage: marked = parse_tolerance("sage: s.update(tol = 0.1); s.rel_tol # abs tol   ␣
↪ 0.01 ", "")
sage: marked.tol
0
sage: marked.rel_tol
0
sage: marked.abs_tol
0.010000000000000000000...?
```

sage.doctest.parsing.**pre_hash**(*s*)

Prepends a string with its length.

EXAMPLES:

```
sage: from sage.doctest.parsing import pre_hash
sage: pre_hash("abc")
'3:abc'
```

sage.doctest.parsing.**reduce_hex**(*fingerprints*)

Return a symmetric function of the arguments as hex strings.

The arguments should be 32 character strings consisting of hex digits: 0-9 and a-f.

EXAMPLES:

```
sage: from sage.doctest.parsing import reduce_hex
sage: reduce_hex(["abc", "12399aedf"])
'00000000000000000000000012399a463'
sage: reduce_hex(["12399aedf","abc"])
'00000000000000000000000012399a463'
```

sage.doctest.parsing.**unparse_optional_tags**(*tags*, *prefix='# '*)

Return a comment string that sets `tags`.

INPUT:

- `tags` – dictionary or iterable of tags, as output by *parse_optional_tags()*
- `prefix` – to be put before a nonempty string

EXAMPLES:

```
sage: from sage.doctest.parsing import unparse_optional_tags
sage: unparse_optional_tags({})
''
sage: unparse_optional_tags({'magma': None})
'# optional - magma'
sage: unparse_optional_tags({'fictional_optional': None,
....:                        'sage.rings.number_field': None,
....:                        'scipy': 'just because',
....:                        'bliss': None})
'# optional - bliss fictional_optional, needs scipy (just because) sage.rings.
↪number_field'
sage: unparse_optional_tags(['long time', 'not tested', 'p4cka9e'], prefix='')
'long time, not tested, optional - p4cka9e'
```

sage.doctest.parsing.**update_optional_tags**(*line*, *tags*, *add_tags*, *remove_tags*, *force_rewrite=None*)

Return the doctest `line` with tags changed.

EXAMPLES:

```
sage: from sage.doctest.parsing import update_optional_tags, optional_tag_columns,
↪ standard_tag_columns
sage: ruler = ''
sage: for column in optional_tag_columns:
....:     ruler += ' ' * (column - len(ruler)) + 'V'
sage: for column in standard_tag_columns:
....:     ruler += ' ' * (column - len(ruler)) + 'v'
sage: def print_with_ruler(lines):
....:     print('|' + ruler)
....:     for line in lines:
....:         print('|' + line)
sage: print_with_ruler([  # the tags are obscured in the source file to avoid␣
↪relint warnings
....:     update_optional_tags('    sage: something()  # opt' 'ional - latte_int',
....:                          remove_tags=['latte_int', 'wasnt_even_there']),
....:     update_optional_tags('    sage: nothing_to_be_seen_here()',
....:                          tags=['scipy', 'long time']),
....:     update_optional_tags('    sage: nothing_to_be_seen_here(honestly=True)',
....:                          add_tags=['scipy', 'long time']),
....:     update_optional_tags('    sage: nothing_to_be_seen_here(honestly=True,␣
↪very=True)',
```

(continues on next page)

(continued from previous page)

```
....:                             add_tags=['scipy', 'long time']),
....:     update_optional_tags('    sage: no_there_is_absolutely_nothing_to_be_
↪seen_here_i_am_serious()#opt' 'ional:bliss',
....:                             add_tags=['scipy', 'long time']),
....:     update_optional_tags('    sage: ntbsh()  # abbrv for above#opt'
↪'ional:bliss',
....:                             add_tags={'scipy': None, 'long time': '30s on the␣
↪highest setting'}),
....:     update_optional_tags('    sage: no_there_is_absolutely_nothing_to_be_
↪seen_here_i_am_serious()  # really, you can trust me here',
....:                             add_tags=['scipy']),
....: ])
|                                                  V       V       V       V      ␣
↪V   V   v           v                   v                                        ␣
↪v
|     sage: something()
|     sage: nothing_to_be_seen_here()              # long time                      ␣
↪       # needs scipy
|     sage: nothing_to_be_seen_here(honestly=True)        # long time              ␣
↪       # needs scipy
|     sage: nothing_to_be_seen_here(honestly=True, very=True)     # long time     ␣
↪       # needs scipy
|     sage: no_there_is_absolutely_nothing_to_be_seen_here_i_am_serious()
↪# long time, optional - bliss, needs scipy
|     sage: ntbsh()  # abbrv for above             # long time (30s on the highest␣
↪setting), optional - bliss, needs scipy
|     sage: no_there_is_absolutely_nothing_to_be_seen_here_i_am_serious()  #␣
↪really, you can trust me here                # needs scipy
```

When no tags are changed, by default, the unchanged input is returned. We can force a rewrite; unconditionally or whenever standard tags are involved. But even when forced, if comments are already aligned at one of the standard alignment columns, this alignment is kept even if we would normally realign farther to the left:

```
sage: print_with_ruler([
....:     update_optional_tags('    sage: unforced()       # opt' 'ional - latte_
↪int'),
....:     update_optional_tags('    sage: unforced()  # opt' 'ional - latte_int',
....:                             add_tags=['latte_int']),
....:     update_optional_tags('    sage: forced()#opt' 'ional- latte_int',
....:                             force_rewrite=True),
....:     update_optional_tags('    sage: forced()  # opt' 'ional - scipy',
....:                             force_rewrite='standard'),
....:     update_optional_tags('    sage: aligned_with_below()                   ␣
↪              # opt' 'ional - 4ti2',
....:                             force_rewrite=True),
....:     update_optional_tags('    sage: aligned_with_above()                   ␣
↪              # opt' 'ional - 4ti2',
....:                             force_rewrite=True),
....:     update_optional_tags('    sage: also_already_aligned()                 ␣
↪                                                                  # ne'
↪'eds scipy',
....:                             force_rewrite='standard'),
....:     update_optional_tags('    sage: two_columns_first_preserved()         #␣
↪lo' 'ng time                              # ne' 'eds scipy',
....:                             force_rewrite='standard'),
....:     update_optional_tags('    sage: two_columns_first_preserved()          ␣
```

(continues on next page)

(continued from previous page)

```
↪      # lo' 'ng time                                    # ne' 'eds scipy',
....:                           force_rewrite='standard'),
....: ])
|                                                  V       V       V       V      ␣
↪V   V   v           v                   v                                        ␣
↪v
|     sage: unforced()       # optional - latte_int
|     sage: unforced()  # optional - latte_int
|     sage: forced()                                 # optional - latte_int
|     sage: forced()                                                              ␣
↪       # needs scipy
|     sage: aligned_with_below()                                    # optional - 4ti2
|     sage: aligned_with_above()                                    # optional - 4ti2
|     sage: also_already_aligned()                                                ␣
↪                                      # needs scipy
|     sage: two_columns_first_preserved()          # long time                    ␣
↪       # needs scipy
|     sage: two_columns_first_preserved()                 # long time             ␣
↪       # needs scipy
```

Rewriting a persistent (block-scoped) tag:

```
sage: print_with_ruler([
....:     update_optional_tags('    sage:    #opt' 'ional:magma sage.symbolic',
....:                          force_rewrite=True),
....: ])
|                                                  V       V       V       V      ␣
↪V   V   v           v                   v                                        ␣
↪v
|     sage: # optional - magma, needs sage.symbolic
```

CHAPTER

FIVE

REPORTING DOCTEST RESULTS

This module determines how doctest results are reported to the user.

It also computes the exit status in the `error_status` attribute of *DocTestReporter*. This is a bitwise OR of the following bits:

- 1: Doctest failure
- 2: Bad command line syntax or invalid options
- 4: Test timed out
- 8: Test exited with nonzero status
- 16: Test crashed with a signal (e.g. segmentation fault)
- 32: TAB character found
- 64: Internal error in the doctesting framework
- 128: Testing interrupted, not all tests run
- 256: Doctest contains explicit source line number

AUTHORS:

- David Roe (2012-03-27) – initial version, based on Robert Bradshaw's code.

class sage.doctest.reporting.**DocTestReporter**(*controller*)

Bases: SageObject

This class reports to the users on the results of doctests.

finalize()

Print out the postscript that summarizes the doctests that were run.

EXAMPLES:

First we have to set up a bunch of stuff:

```
sage: from sage.doctest.reporting import DocTestReporter
sage: from sage.doctest.control import DocTestController, DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource, DictAsObject
sage: from sage.doctest.forker import SageDocTestRunner
sage: from sage.doctest.parsing import SageOutputChecker
sage: from sage.doctest.util import Timer
sage: import doctest
sage: filename = sage.doctest.reporting.__file__
sage: DD = DocTestDefaults()
sage: FDS = FileDocTestSource(filename, DD)
```

(continues on next page)

(continued from previous page)

```
sage: DC = DocTestController(DD, [filename])
sage: DTR = DocTestReporter(DC)
```

Now we pretend to run some doctests:

```
sage: DTR.report(FDS, True, 0, None, "Output so far...", pid=1234)
    Timed out
**********************************************************************
Tests run before process (pid=1234) timed out:
Output so far...
**********************************************************************
sage: DTR.report(FDS, False, 3, None, "Output before bad exit")
    Bad exit: 3
**********************************************************************
Tests run before process failed:
Output before bad exit
**********************************************************************
sage: doctests, extras = FDS.create_doctests(globals())
sage: runner = SageDocTestRunner(
....:     SageOutputChecker(), verbose=False, sage_options=DD,
....:     optionflags=doctest.NORMALIZE_WHITESPACE|doctest.ELLIPSIS)
sage: t = Timer().start().stop()
sage: t.annotate(runner)
sage: DC.timer = t
sage: D = DictAsObject({'err':None})
sage: runner.update_results(D)
0
sage: DTR.report(FDS, False, 0, (sum([len(t.examples) for t in doctests]), D),
....:            "Good tests")
    [... tests, ...s wall]
sage: runner.failures = 1
sage: runner.update_results(D)
1
sage: DTR.report(FDS, False, 0, (sum([len(t.examples) for t in doctests]), D),
....:            "Doctest output including the failure...")
    [... tests, 1 failure, ...s wall]
```

Now we can show the output of finalize:

```
sage: DC.sources = [None] * 4 # to fool the finalize method
sage: DTR.finalize()
----------------------------------------------------------------------
sage -t .../sage/doctest/reporting.py  # Timed out
sage -t .../sage/doctest/reporting.py  # Bad exit: 3
sage -t .../sage/doctest/reporting.py  # 1 doctest failed
----------------------------------------------------------------------
Total time for all tests: 0.0 seconds
    cpu time: 0.0 seconds
    cumulative wall time: 0.0 seconds
```

If we interrupted doctests, then the number of files tested will not match the number of sources on the controller:

```
sage: DC.sources = [None] * 6
sage: DTR.finalize()

----------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
sage -t .../sage/doctest/reporting.py  # Timed out
sage -t .../sage/doctest/reporting.py  # Bad exit: 3
sage -t .../sage/doctest/reporting.py  # 1 doctest failed
Doctests interrupted: 4/6 files tested
----------------------------------------------------------------------
Total time for all tests: 0.0 seconds
    cpu time: 0.0 seconds
    cumulative wall time: 0.0 seconds
```

report (*source*, *timeout*, *return_code*, *results*, *output*, *pid=None*)

Report on the result of running doctests on a given source.

This doesn't print the *report_head()*, which is assumed to be printed already.

INPUT:

- `source` – a source from *sage.doctest.sources*
- `timeout` – boolean; whether doctests timed out
- `return_code` – integer; the return code of the process running doctests on that file
- `results` – (irrelevant if `timeout` or `return_code`) a tuple
  - `ntests` – the number of doctests
  - `timings` – a *sage.doctest.sources.DictAsObject* instance storing timing data
- `output` – string; printed if there was some kind of failure
- `pid` – integer (default: `None`); the pid of the worker process

EXAMPLES:

```
sage: from sage.doctest.reporting import DocTestReporter
sage: from sage.doctest.control import DocTestController, DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource, DictAsObject
sage: from sage.doctest.forker import SageDocTestRunner
sage: from sage.doctest.parsing import SageOutputChecker
sage: from sage.doctest.util import Timer
sage: import doctest
sage: filename = sage.doctest.reporting.__file__
sage: DD = DocTestDefaults()
sage: FDS = FileDocTestSource(filename, DD)
sage: DC = DocTestController(DD, [filename])
sage: DTR = DocTestReporter(DC)
```

You can report a timeout:

```
sage: DTR.report(FDS, True, 0, None, "Output so far...", pid=1234)
    Timed out
**********************************************************************
Tests run before process (pid=1234) timed out:
Output so far...
**********************************************************************
sage: DTR.stats
{'sage.doctest.reporting': {'failed': True,
                            'ntests': 0,
                            'walltime': 1000000.0}}
```

Or a process that returned a bad exit code:

```
sage: DTR.report(FDS, False, 3, None, "Output before trouble")
    Bad exit: 3
**********************************************************************
Tests run before process failed:
Output before trouble
**********************************************************************
sage: DTR.stats
{'sage.doctest.reporting': {'failed': True,
                            'ntests': 0,
                            'walltime': 1000000.0}}
```

Or a process that segfaulted:

```
sage: from signal import SIGSEGV
sage: DTR.report(FDS, False, -SIGSEGV, None, "Output before trouble")
    Killed due to segmentation fault
**********************************************************************
Tests run before process failed:
Output before trouble
**********************************************************************
sage: DTR.stats
{'sage.doctest.reporting': {'failed': True,
                            'ntests': 0,
                            'walltime': 1000000.0}}
```

Report a timeout with results and a `SIGKILL`:

```
sage: from signal import SIGKILL
sage: DTR.report(FDS, True, -SIGKILL, (1,None), "Output before trouble")
    Timed out after testing finished (and interrupt failed)
**********************************************************************
Tests run before process timed out:
Output before trouble
**********************************************************************
sage: DTR.stats
{'sage.doctest.reporting': {'failed': True,
                            'ntests': 1,
                            'walltime': 1000000.0}}
```

This is an internal error since results is None:

```
sage: DTR.report(FDS, False, 0, None, "All output")
    Error in doctesting framework (bad result returned)
**********************************************************************
Tests run before error:
All output
**********************************************************************
sage: DTR.stats
{'sage.doctest.reporting': {'failed': True,
                            'ntests': 1,
                            'walltime': 1000000.0}}
```

Or tell the user that everything succeeded:

```
sage: doctests, extras = FDS.create_doctests(globals())
sage: runner = SageDocTestRunner(
....:     SageOutputChecker(), verbose=False, sage_options=DD,
```

(continues on next page)

(continued from previous page)

```
....:     optionflags=doctest.NORMALIZE_WHITESPACE|doctest.ELLIPSIS)
sage: Timer().start().stop().annotate(runner)
sage: D = DictAsObject({'err':None})
sage: runner.update_results(D)
0
sage: DTR.report(FDS, False, 0, (sum([len(t.examples) for t in doctests]), D),
....:            "Good tests")
    [... tests, ...s wall]
sage: DTR.stats
{'sage.doctest.reporting': {'ntests': ..., 'walltime': ...}}
```

Or inform the user that some doctests failed:

```
sage: runner.failures = 1
sage: runner.update_results(D)
1
sage: DTR.report(FDS, False, 0, (sum([len(t.examples) for t in doctests]), D),
....:            "Doctest output including the failure...")
    [... tests, 1 failure, ...s wall]
```

If the user has requested that we report on skipped doctests, we do so:

```
sage: DC.options = DocTestDefaults(show_skipped=True)
sage: from collections import defaultdict
sage: optionals = defaultdict(int)
sage: optionals['magma'] = 5; optionals['long time'] = 4; optionals[''] = 1;␣
↪optionals['not tested'] = 2
sage: D = DictAsObject(dict(err=None, optionals=optionals))
sage: runner.failures = 0
sage: runner.update_results(D)
0
sage: DTR.report(FDS, False, 0, (sum([len(t.examples) for t in doctests]), D),
↪ "Good tests")
    1 unlabeled test not run
    4 long tests not run
    5 magma tests not run
    2 not tested tests not run
    0 tests not run because we ran out of time
    [... tests, ...s wall]
```

Test an internal error in the reporter:

```
sage: DTR.report(None, None, None, None, None)
Traceback (most recent call last):
...
AttributeError: 'NoneType' object has no attribute 'basename'...
```

The only-errors mode does not output anything on success:

```
sage: DD = DocTestDefaults(only_errors=True)
sage: FDS = FileDocTestSource(filename, DD)
sage: DC = DocTestController(DD, [filename])
sage: DTR = DocTestReporter(DC)
sage: doctests, extras = FDS.create_doctests(globals())
sage: runner = SageDocTestRunner(
....:     SageOutputChecker(), verbose=False, sage_options=DD,
....:     optionflags=doctest.NORMALIZE_WHITESPACE|doctest.ELLIPSIS)
```

(continues on next page)

(continued from previous page)

```
sage: Timer().start().stop().annotate(runner)
sage: D = DictAsObject({'err':None})
sage: runner.update_results(D)
0
sage: DTR.report(FDS, False, 0, (sum([len(t.examples) for t in doctests]), D),
....:            "Good tests")
```

However, failures are still output in the errors-only mode:

```
sage: runner.failures = 1
sage: runner.update_results(D)
1
sage: DTR.report(FDS, False, 0, (sum([len(t.examples) for t in doctests]), D),
....:            "Failed test")
    [... tests, 1 failure, ...s wall]
```

report_head(*source*, *fail_msg=None*)

Return the `sage -t [options] file.py` line as string.

INPUT:

- `source` – a source from *sage.doctest.sources*
- `fail_msg` – `None` or a string

EXAMPLES:

```
sage: from sage.doctest.reporting import DocTestReporter
sage: from sage.doctest.control import DocTestController, DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.doctest.forker import SageDocTestRunner
sage: filename = sage.doctest.reporting.__file__
sage: DD = DocTestDefaults()
sage: FDS = FileDocTestSource(filename, DD)
sage: DC = DocTestController(DD, [filename])
sage: DTR = DocTestReporter(DC)
sage: print(DTR.report_head(FDS))
sage -t .../sage/doctest/reporting.py
```

The same with various options:

```
sage: DD.long = True
sage: print(DTR.report_head(FDS))
sage -t --long .../sage/doctest/reporting.py
sage: print(DTR.report_head(FDS, "Failed by self-sabotage"))
sage -t --long .../sage/doctest/reporting.py  # Failed by self-sabotage
```

were_doctests_with_optional_tag_run(*tag*)

Return whether doctests marked with this tag were run.

INPUT:

- `tag` – string

EXAMPLES:

```
sage: from sage.doctest.reporting import DocTestReporter
sage: from sage.doctest.control import DocTestController, DocTestDefaults
```

(continues on next page)

(continued from previous page)

```
sage: filename = sage.doctest.reporting.__file__
sage: DC = DocTestController(DocTestDefaults(), [filename])
sage: DTR = DocTestReporter(DC)
```

```
sage: DTR.were_doctests_with_optional_tag_run('sage')
True
sage: DTR.were_doctests_with_optional_tag_run('nice_unavailable_package')
False
```

When latex is available, doctests marked with optional tag `latex` are run by default since Issue #32174:

```
sage: # needs SAGE_SRC
sage: filename = os.path.join(SAGE_SRC, 'sage', 'misc', 'latex.py')
sage: DC = DocTestController(DocTestDefaults(), [filename])
sage: DTR = DocTestReporter(DC)
sage: DTR.were_doctests_with_optional_tag_run('latex')   # optional - latex
True
```

sage.doctest.reporting.**signal_name**(*sig*)

Return a string describing a signal number.

EXAMPLES:

```
sage: from signal import SIGSEGV
sage: from sage.doctest.reporting import signal_name
sage: signal_name(SIGSEGV)
'segmentation fault'
sage: signal_name(9)
'kill signal'
sage: signal_name(12345)
'signal 12345'
```

CHAPTER

SIX

DETECTING EXTERNAL SOFTWARE

This module makes up a list of external software that Sage interfaces. Availability of each software is tested only when necessary. This is mainly used for the doctests which require certain external software installed on the system.

Even though the functions in this module should also work when an external software is not present, most doctests in this module are only tested if testing of external software is explicitly enabled in order to avoid invoking external software otherwise. See Issue #28819 for details.

AUTHORS:

- Kwankyu Lee (2016-03-09) – initial version, based on code by Robert Bradshaw and Nathann Cohen

class sage.doctest.external.**AvailableSoftware**

Bases: object

This class keeps the set of available software whose availability is detected lazily from the list of external software.

EXAMPLES:

```
sage: from sage.doctest.external import external_software, available_software
sage: external_software
['cplex',
 'dvips',
 'ffmpeg',
 'gurobi',
 'internet',
 'latex',
 'latex_package_tkz_graph',
 'lualatex',
 'macaulay2',
 'magma',
 'maple',
 'mathematica',
 'matlab',
 'octave',
 'pdflatex',
 'scilab',
 'xelatex']
sage: 'internet' in available_software # random, optional - internet
True
sage: available_software.issuperset(set(['internet','latex'])) # random, optional␣
↪- internet latex
True
```

detectable()

Return the list of names of those features for which testing their presence is allowed.

hidden()

Return the list of detected hidden external software.

EXAMPLES:

```
sage: # needs conway_polynomials database_cremona_mini_ellcurve database_
↪ellcurves database_graphs
sage: from sage.doctest.external import available_software
sage: from sage.features.databases import all_features
sage: for f in all_features():
....:     f.hide()
....:     if f._spkg_type() == 'standard':
....:         test = f.name in available_software
....:     f.unhide()
sage: sorted(available_software.hidden())
[...'conway_polynomials',...
 'database_cremona_mini_ellcurve',...
 'database_ellcurves',...
 'database_graphs'...]
```

issuperset(*other*)

Return `True` if `other` is a subset of `self`.

EXAMPLES:

```
sage: from sage.doctest.external import available_software
sage: available_software.issuperset(set(['internet','latex','magma'])) #␣
↪random, optional - internet latex magma
True
```

seen()

Return the list of detected external software.

EXAMPLES:

```
sage: from sage.doctest.external import available_software
sage: available_software.seen() # random
['internet', 'latex', 'magma']
```

sage.doctest.external.**external_features**()

Generate the features that are only to be tested if `--optional=external` is used.

EXAMPLES:

```
sage: from sage.doctest.external import external_features
sage: next(external_features())
Feature('internet')
```

sage.doctest.external.**has_4ti2**()

Test if the 4ti2 package is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_4ti2
sage: has_4ti2()    # optional -- 4ti2
FeatureTestResult('4ti2', True)
```

sage.doctest.external.**has_cplex**()

Test if CPLEX is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_cplex
sage: has_cplex() # random, optional - CPLEX
FeatureTestResult('cplex', True)
```

sage.doctest.external.**has_dvipng**()

Test if dvipng is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_dvipng
sage: has_dvipng() # optional -- dvipng
FeatureTestResult('dvipng', True)
```

sage.doctest.external.**has_ffmpeg**()

Test if ffmpeg is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_ffmpeg
sage: has_ffmpeg()      # optional -- ffmpeg
FeatureTestResult('ffmpeg', True)
```

sage.doctest.external.**has_graphviz**()

Test if graphviz (dot, twopi, neato) are available.

EXAMPLES:

```
sage: from sage.doctest.external import has_graphviz
sage: has_graphviz()    # optional -- graphviz
FeatureTestResult('graphviz', True)
```

sage.doctest.external.**has_gurobi**()

Test if Gurobi is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_gurobi
sage: has_gurobi() # random, optional - Gurobi
FeatureTestResult('gurobi', True)
```

sage.doctest.external.**has_imagemagick**()

Test if ImageMagick (command magick or convert) is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_imagemagick
sage: has_imagemagick() # optional -- imagemagick
FeatureTestResult('imagemagick', True)
```

sage.doctest.external.**has_internet**()

Test if Internet is available.

Failure of connecting to the site "https://www.sagemath.org" within a second is regarded as internet being not available.

EXAMPLES:

```
sage: from sage.doctest.external import has_internet
sage: has_internet()  # random, optional -- internet
FeatureTestResult('internet', True)
```

sage.doctest.external.**has_latex**()

Test if Latex is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_latex
sage: has_latex() # optional - latex
FeatureTestResult('latex', True)
```

sage.doctest.external.**has_lualatex**()

Test if lualatex is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_lualatex
sage: has_lualatex()   # optional - lualatex
FeatureTestResult('lualatex', True)
```

sage.doctest.external.**has_macaulay2**()

Test if Macaulay2 is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_macaulay2
sage: has_macaulay2() # random, optional - macaulay2
True
```

sage.doctest.external.**has_magma**()

Test if Magma is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_magma
sage: has_magma() # random, optional - magma
True
```

sage.doctest.external.**has_maple**()

Test if Maple is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_maple
sage: has_maple() # random, optional - maple
True
```

sage.doctest.external.**has_mathematica**()

Test if Mathematica is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_mathematica
sage: has_mathematica() # random, optional - mathematica
True
```

sage.doctest.external.**has_matlab**()

Test if Matlab is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_matlab
sage: has_matlab() # random, optional - matlab
True
```

sage.doctest.external.**has_octave**()

Test if Octave is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_octave
sage: has_octave() # random, optional - octave
True
```

sage.doctest.external.**has_pandoc**()

Test if pandoc is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_pandoc
sage: has_pandoc()      # optional -- pandoc
FeatureTestResult('pandoc', True)
```

sage.doctest.external.**has_pdf2svg**()

Test if pdf2svg is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_pdf2svg
sage: has_pdf2svg() # optional -- pdf2svg
FeatureTestResult('pdf2svg', True)
```

sage.doctest.external.**has_pdflatex**()

Test if pdflatex is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_pdflatex
sage: has_pdflatex()    # optional - pdflatex
FeatureTestResult('pdflatex', True)
```

sage.doctest.external.**has_rubiks**()

Test if the rubiks package (`cu2`, `cubex`, `dikcube`, `mcube`, `optimal`, and `size222`) is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_rubiks
sage: has_rubiks()    # optional -- rubiks
FeatureTestResult('rubiks', True)
```

sage.doctest.external.**has_scilab**()

Test if Scilab is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_scilab
sage: has_scilab() # random, optional - scilab
True
```

sage.doctest.external.**has_xelatex**()

Test if xelatex is available.

EXAMPLES:

```
sage: from sage.doctest.external import has_xelatex
sage: has_xelatex()   # optional - xelatex
FeatureTestResult('xelatex', True)
```

CHAPTER

SEVEN

TEST THE DOCTESTING FRAMEWORK

Many tests (with expected failures or crashes) are run in a subprocess, those tests can be found in the `tests/` subdirectory.

EXAMPLES:

```
sage: import signal
sage: import subprocess
sage: import time
sage: from sage.env import SAGE_SRC
sage: tests_dir = os.path.join(SAGE_SRC, 'sage', 'doctest', 'tests')
sage: tests_env = dict(os.environ)
```

Unset `TERM` when running doctests, see Issue #14370:

```
sage: try:
....:     del tests_env['TERM']
....: except KeyError:
....:     pass
sage: kwds = {'cwd': tests_dir, 'env':tests_env}
```

Check that Issue #2235 has been fixed:

```
sage: subprocess.call(["sage", "-t", "--warn-long", "0",     # long time
....:       "--random-seed=0", "--optional=sage", "longtime.rst"], **kwds)
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 --random-seed=0 longtime.rst
[0 tests, ...s wall]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
...
0
sage: subprocess.call(["sage", "-t", "--warn-long", "0",     # long time
....:      "--random-seed=0", "--optional=sage", "-l", "longtime.rst"], **kwds)
Running doctests...
Doctesting 1 file.
sage -t --long --warn-long 0.0 --random-seed=0 longtime.rst
[1 test, ...s wall]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
...
0
```

Check handling of tolerances:

```
sage: subprocess.call(["sage", "-t", "--warn-long", "0",     # long time
....:       "--random-seed=0", "--optional=sage", "tolerance.rst"], **kwds)
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 --random-seed=0 tolerance.rst
**********************************************************************
File "tolerance.rst", line ..., in sage.doctest.tests.tolerance
Failed example:
    print(":-(")    # abs tol 0.1
Expected:
    :-)
Got:
    :-(
**********************************************************************
File "tolerance.rst", line ..., in sage.doctest.tests.tolerance
Failed example:
    print("1.0 2.0 3.0")  # abs tol 0.1
Expected:
    4.0 5.0
Got:
    1.0 2.0 3.0
**********************************************************************
File "tolerance.rst", line ..., in sage.doctest.tests.tolerance
Failed example:
    print("Hello")  # abs tol 0.1
Expected:
    1.0
Got:
    Hello
**********************************************************************
File "tolerance.rst", line ..., in sage.doctest.tests.tolerance
Failed example:
    print("1.0")  # abs tol 0.1
Expected:
    Hello
Got:
    1.0
**********************************************************************
File "tolerance.rst", line ..., in sage.doctest.tests.tolerance
Failed example:
    print("Hello 1.1")  # abs tol 0.1
Expected:
    Goodbye 1.0
Got:
    Hello 1.1
**********************************************************************
File "tolerance.rst", line ..., in sage.doctest.tests.tolerance
Failed example:
    print("Hello 1.0")  # rel tol 1e-6
Expected:
    Goodbye 0.999999
Got:
    Hello 1.0
Tolerance exceeded:
    0.999999 vs 1.0, tolerance 2e-6 > 1e-6
**********************************************************************
```

(continues on next page)

(continued from previous page)

```
File "tolerance.rst", line ..., in sage.doctest.tests.tolerance
Failed example:
    print("Hello 1.0")  # rel tol 1e-6
Expected:
    Hello ...
Got:
    Hello 1.0
Note: combining tolerance (# tol) with ellipsis (...) is not supported
**********************************************************************
...
1
```

Test the `--initial` option:

```
sage: subprocess.call(["sage", "-t", "--warn-long", "0",     # long time
....:       "--random-seed=0", "--optional=sage", "-i", "initial.rst"], **kwds)
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 --random-seed=0 initial.rst
**********************************************************************
File "initial.rst", line 4, in sage.doctest.tests.initial
Failed example:
    a = binomiak(10,5)  # random to test that we still get the exception
Exception raised:
    Traceback (most recent call last):
    ...
    NameError: name 'binomiak' is not defined
**********************************************************************
File "initial.rst", line 14, in sage.doctest.tests.initial
Failed example:
    binomial(10,5)
Expected:
    255
Got:
    252
**********************************************************************
...
----------------------------------------------------------------------
sage -t  --warn-long 0.0 --random-seed=0 initial.rst  # 5 doctests failed
----------------------------------------------------------------------
...
1
```

Test the `--exitfirst` option:

```
sage: subprocess.call(["sage", "-t", "--warn-long", "0",     # long time
....:       "--random-seed=0", "--optional=sage", "--exitfirst", "initial.rst"],␣
↪**kwds)
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 --random-seed=0 initial.rst
**********************************************************************
File "initial.rst", line 4, in sage.doctest.tests.initial
Failed example:
    a = binomiak(10,5)  # random to test that we still get the exception
Exception raised:
    Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
    ...
    NameError: name 'binomiak' is not defined
**********************************************************************
...
----------------------------------------------------------------------
sage -t  --warn-long 0.0 --random-seed=0 initial.rst  # 1 doctest failed
----------------------------------------------------------------------
...
1
```

Test a timeout using the `SAGE_TIMEOUT` environment variable. Also set `CYSIGNALS_CRASH_NDEBUG` to help ensure the test times out in a timely manner (Issue #26912):

```
sage: from copy import deepcopy
sage: kwds2 = deepcopy(kwds)
sage: kwds2['env'].update({'SAGE_TIMEOUT': '1', 'CYSIGNALS_CRASH_NDEBUG': '1'})
sage: subprocess.call(["sage", "-t", "--warn-long", "0",     # long time
....:       "--random-seed=0", "--optional=sage", "99seconds.rst"], **kwds2)
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 --random-seed=0 99seconds.rst
    Timed out
**********************************************************************
Tests run before process (pid=...) timed out:
...
----------------------------------------------------------------------
sage -t --warn-long 0.0 --random-seed=0 99seconds.rst  # Timed out
----------------------------------------------------------------------
...
4
```

Test handling of `KeyboardInterrupt` in doctests:

```
sage: subprocess.call(["sage", "-t", "--warn-long", "0",     # long time
....:       "--random-seed=0", "--optional=sage", "keyboardinterrupt.rst"], **kwds)
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 --random-seed=0 keyboardinterrupt.rst
**********************************************************************
File "keyboardinterrupt.rst", line 11, in sage.doctest.tests.keyboardinterrupt
Failed example:
    raise KeyboardInterrupt
Exception raised:
    Traceback (most recent call last):
    ...
    KeyboardInterrupt
**********************************************************************
...
----------------------------------------------------------------------
sage -t --warn-long 0.0 --random-seed=0 keyboardinterrupt.rst  # 1 doctest failed
----------------------------------------------------------------------
...
1
```

Interrupt the doctester:

```
sage: subprocess.call(["sage", "-t", "--warn-long", "0",     # long time
....:       "--random-seed=0", "--optional=sage", "interrupt.rst"], **kwds)
Running doctests...
Doctesting 1 file.
Killing test interrupt.rst
----------------------------------------------------------------------
Doctests interrupted: 0/1 files tested
----------------------------------------------------------------------
...
128
```

Interrupt the doctester (while parallel testing) when a doctest cannot be interrupted. We also test that passing a ridiculous number of threads doesn't hurt:

```
sage: F = tmp_filename()
sage: from copy import deepcopy
sage: kwds2 = deepcopy(kwds)
sage: kwds2['env']['DOCTEST_TEST_PID_FILE'] = F  # Doctester will write its PID in␣
↪this file
sage: subprocess.call(["sage", "-tp", "1000000", "--timeout=120",  # long time
....:       "--die_timeout=10", "--optional=sage",
....:       "--warn-long", "0", "99seconds.rst", "interrupt_diehard.rst"], **kwds2)
Running doctests...
Doctesting 2 files using 1000000 threads...
Killing test 99seconds.rst
Killing test interrupt_diehard.rst
----------------------------------------------------------------------
Doctests interrupted: 0/2 files tested
----------------------------------------------------------------------
...
128
```

Even though the doctester master process has exited, the child process is still alive, but it should be killed automatically after the `die_timeout` given above (10 seconds):

```
sage: # long time
sage: pid = int(open(F).read())
sage: time.sleep(2)
sage: os.kill(pid, signal.SIGQUIT)  # 2 seconds passed => still alive
sage: time.sleep(8)
sage: os.kill(pid, signal.SIGQUIT)  # 10 seconds passed => dead  # random
Traceback (most recent call last):
...
ProcessLookupError: ...
```

If the child process is dead and removed, the last output should be as above. However, the child process interrupted its parent process (see `'interrupt_diehard.rst'`), and became an orphan process. Depending on the system, an orphan process may eventually become a zombie process instead of being removed, and then the last output would just be a blank. Hence the `# random` tag.

Test a doctest failing with `abort()`:

```
sage: subprocess.call(["sage", "-t", "--warn-long", "0",     # long time
....:       "--random-seed=0", "--optional=sage", "abort.rst"], **kwds)
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 --random-seed=0 abort.rst
```

(continues on next page)

(continued from previous page)

```
    Killed due to abort
**********************************************************************
Tests run before process (pid=...) failed:
...
----------------------------------------------------------------------
Unhandled SIGABRT: An abort() occurred.
This probably occurred because a *compiled* module has a bug
in it and is not properly wrapped with sig_on(), sig_off().
Python will now terminate.
----------------------------------------------------------------------
...
----------------------------------------------------------------------
sage -t --warn-long 0.0 --random-seed=0 abort.rst  # Killed due to abort
----------------------------------------------------------------------
...
16
```

A different kind of crash:

```
sage: subprocess.call(["sage", "-t", "--warn-long", "0",     # long time
....:       "--random-seed=0", "--optional=sage", "fail_and_die.rst"], **kwds)
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 --random-seed=0 fail_and_die.rst
**********************************************************************
File "fail_and_die.rst", line 5, in sage.doctest.tests.fail_and_die
Failed example:
    this_gives_a_NameError
Exception raised:
    Traceback (most recent call last):
    ...
    NameError: name 'this_gives_a_NameError' is not defined
    Killed due to kill signal
**********************************************************************
Tests run before process (pid=...) failed:
...
----------------------------------------------------------------------
sage -t --warn-long 0.0 --random-seed=0 fail_and_die.rst  # Killed due to kill signal
----------------------------------------------------------------------
...
16
```

Test that `sig_on_count` is checked correctly:

```
sage: subprocess.call(["sage", "-t", "--warn-long", "0",     # long time
....:       "--random-seed=0", "--optional=sage", "sig_on.rst"], **kwds)
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 --random-seed=0 sig_on.rst
**********************************************************************
File "sig_on.rst", line 6, in sage.doctest.tests.sig_on
Failed example:
    sig_on_count() # check sig_on/off pairings (virtual doctest)
Expected:
    0
Got:
    1
```

(continues on next page)

(continued from previous page)

```
**********************************************************************
1 item had failures:
   1 of   5 in sage.doctest.tests.sig_on
    [3 tests, 1 failure, ...]
----------------------------------------------------------------------
sage -t --warn-long 0.0 --random-seed=0 sig_on.rst  # 1 doctest failed
----------------------------------------------------------------------
...
1
```

Test logfiles in serial and parallel mode (see Issue #19271):

```
sage: t = tmp_filename()
sage: subprocess.call(["sage", "-t", "--serial", "--warn-long", "0",    # long time
....:       "--random-seed=0", "--optional=sage", "--logfile", t, "simple_failure.rst
↪"],
....:       stdout=open(os.devnull, "w"), **kwds)
1
sage: print(open(t).read())  # long time
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 --random-seed=0 simple_failure.rst
**********************************************************************
File "simple_failure.rst", line 7, in sage.doctest.tests.simple_failure
Failed example:
    a * b
Expected:
    20
Got:
    15
**********************************************************************
1 item had failures:
   1 of   5 in sage.doctest.tests.simple_failure
    [4 tests, 1 failure, ...]
----------------------------------------------------------------------
sage -t --warn-long 0.0 --random-seed=0 simple_failure.rst  # 1 doctest failed
----------------------------------------------------------------------
...

sage: subprocess.call(["sage", "-t", "--warn-long", "0",    # long time
....:       "--random-seed=0", "--optional=sage", "--logfile", t, "simple_failure.rst
↪"],
....:       stdout=open(os.devnull, "w"), **kwds)
1
sage: print(open(t).read())  # long time
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 --random-seed=0 simple_failure.rst
**********************************************************************
File "simple_failure.rst", line 7, in sage.doctest.tests.simple_failure
Failed example:
    a * b
Expected:
    20
Got:
    15
**********************************************************************
```

(continues on next page)

(continued from previous page)

```
1 item had failures:
   1 of   5 in sage.doctest.tests.simple_failure
    [4 tests, 1 failure, ...]
----------------------------------------------------------------------
sage -t --warn-long 0.0 --random-seed=0 simple_failure.rst  # 1 doctest failed
----------------------------------------------------------------------
...
```

Test the `--debug` option:

```
sage: subprocess.call(["sage", "-t", "--warn-long", "0",     # long time
....:       "--random-seed=0", "--optional=sage", "--debug", "simple_failure.rst"],
....:       stdin=open(os.devnull), **kwds)
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 --random-seed=0 simple_failure.rst
**********************************************************************
File "simple_failure.rst", line 7, in sage.doctest.tests.simple_failure
Failed example:
    a * b
Expected:
    20
Got:
    15
**********************************************************************
Previously executed commands:
    s...: a = 3
    s...: b = 5
    s...: a + b
    8
sage:

Returning to doctests...
**********************************************************************
1 item had failures:
   1 of   5 in sage.doctest.tests.simple_failure
    [4 tests, 1 failure, ...]
----------------------------------------------------------------------
sage -t --warn-long 0.0 --random-seed=0 simple_failure.rst  # 1 doctest failed
----------------------------------------------------------------------
...
1
```

Test running under gdb, without and with a timeout:

```
sage: subprocess.call(["sage", "-t",  "--warn-long", "0",    # long time, optional: gdb
....:       "--random-seed=0", "--optional=sage", "--gdb", "1second.rst"],
....:       stdin=open(os.devnull), **kwds)
exec gdb ...
Running doctests...
Doctesting 1 file...
sage -t... 1second.rst...
    [2 tests, ...s wall]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
...
```

(continues on next page)

(continued from previous page)

```
0
```

gdb might need a long time to start up, so we allow 30 seconds:

```
sage: subprocess.call(["sage", "-t",  "--warn-long", "0",    # long time, optional: gdb
....:       "--random-seed=0", "--optional=sage", "--gdb", "-T30", "99seconds.rst"],
....:       stdin=open(os.devnull), **kwds)
exec gdb ...
Running doctests...
    Timed out
4
```

Test the `--show-skipped` option:

```
sage: subprocess.call(["sage", "-t", "--warn-long", "0",    # long time
....:       "--random-seed=0", "--optional=sage", "--show-skipped", "show_skipped.rst
↪"], **kwds)
Running doctests ...
Doctesting 1 file.
sage -t --warn-long 0.0 --random-seed=0 show_skipped.rst
    2 tests not run due to known bugs
    1 gap test not run
    1 long test not run
    1 not tested test not run
    0 tests not run because we ran out of time
    [2 tests, ...s wall]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
...
0
```

Optional tests are run correctly:

```
sage: subprocess.call(["sage", "-t", "--warn-long", "0", "--long",  # long time
....:       "--random-seed=0", "--show-skipped", "--optional=sage,gap", "show_skipped.
↪rst"], **kwds)
Running doctests ...
Doctesting 1 file.
sage -t --long --warn-long 0.0 --random-seed=0 show_skipped.rst
    2 tests not run due to known bugs
    1 not tested test not run
    0 tests not run because we ran out of time
    [4 tests, ...s wall]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
...
0

sage: subprocess.call(["sage", "-t", "--warn-long", "0", "--long",  # long time
....:       "--random-seed=0", "--show-skipped", "--optional=gAp", "show_skipped.rst"],
↪ **kwds)
Running doctests ...
Doctesting 1 file.
sage -t --long --warn-long 0.0 --random-seed=0 show_skipped.rst
    2 tests not run due to known bugs
```

(continues on next page)

(continued from previous page)

```
    1 not tested test not run
    2 sage tests not run
    0 tests not run because we ran out of time
    [2 tests, ...s wall]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
...
0
```

Test an invalid value for `--optional`:

```
sage: subprocess.call(["sage", "-t", "--warn-long", "0",
....:       "--random-seed=0", "--optional=bad-option", "show_skipped.rst"], **kwds)
Traceback (most recent call last):
...
ValueError: invalid optional tag 'bad-option'
1
```

Test `atexit` support in the doctesting framework:

```
sage: F = tmp_filename()
sage: os.path.isfile(F)
True
sage: from copy import deepcopy
sage: kwds2 = deepcopy(kwds)
sage: kwds2['env']['DOCTEST_DELETE_FILE'] = F
sage: subprocess.call(["sage", "-t", "--warn-long", "0",     # long time
....:       "--random-seed=0", "--optional=sage", "atexit.rst"], **kwds2)
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 --random-seed=0 atexit.rst
    [3 tests, ...s wall]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
...
0
sage: os.path.isfile(F)  # long time
False
sage: try:
....:     os.unlink(F)
....: except OSError:
....:     pass
```

Test that random tests are reproducible:

```
sage: subprocess.call(["sage", "-t", "--warn-long", "0",     # long time
....:       "--random-seed=0", "--optional=sage", "random_seed.rst"], **kwds)
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 --random-seed=0 random_seed.rst
**********************************************************************
File "random_seed.rst", line 3, in sage.doctest.tests.random_seed
Failed example:
    randint(5, 10)
Expected:
```

(continues on next page)

(continued from previous page)

```
    9
Got:
    5
**********************************************************************
1 item had failures:
   1 of   2 in sage.doctest.tests.random_seed
    [1 test, 1 failure, ...s wall]
----------------------------------------------------------------------
sage -t --warn-long 0.0 --random-seed=0 random_seed.rst  # 1 doctest failed
----------------------------------------------------------------------
...
1
sage: subprocess.call(["sage", "-t", "--warn-long", "0",     # long time
....:                  "--random-seed=1", "--optional=sage", "random_seed.rst"], **kwds)
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 --random-seed=1 random_seed.rst
    [1 test, ...s wall]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
...
0
```

CHAPTER

EIGHT

UTILITY FUNCTIONS

This module contains various utility functions and classes used in doctesting.

AUTHORS:

- David Roe (2012-03-27) – initial version, based on Robert Bradshaw's code.

class sage.doctest.util.**NestedName**(*base*)

Bases: object

Class used to construct fully qualified names based on indentation level.

EXAMPLES:

```
sage: from sage.doctest.util import NestedName
sage: qname = NestedName('sage.categories.algebras')
sage: qname[0] = 'Algebras'; qname
sage.categories.algebras.Algebras
sage: qname[4] = '__contains__'; qname
sage.categories.algebras.Algebras.__contains__
sage: qname[4] = 'ParentMethods'
sage: qname[8] = 'from_base_ring'; qname
sage.categories.algebras.Algebras.ParentMethods.from_base_ring
```

class sage.doctest.util.**RecordingDict**(**args*, ***kwds*)

Bases: dict

This dictionary is used for tracking the dependencies of an example.

This feature allows examples in different doctests to be grouped for better timing data. It's obtained by recording whenever anything is set or retrieved from this dictionary.

EXAMPLES:

```
sage: from sage.doctest.util import RecordingDict
sage: D = RecordingDict(test=17)
sage: D.got
set()
sage: D['test']
17
sage: D.got
{'test'}
sage: D.set
set()
sage: D['a'] = 1
sage: D['a']
1
```

(continues on next page)

(continued from previous page)

```
sage: D.set
{'a'}
sage: D.got
{'test'}
```

copy()

Note that set and got are not copied.

EXAMPLES:

```
sage: from sage.doctest.util import RecordingDict
sage: D = RecordingDict(d = 42)
sage: D['a'] = 4
sage: D.set
{'a'}
sage: E = D.copy()
sage: E.set
set()
sage: sorted(E.keys())
['a', 'd']
```

get(*name*, *default=None*)

EXAMPLES:

```
sage: from sage.doctest.util import RecordingDict
sage: D = RecordingDict(d = 42)
sage: D.get('d')
42
sage: D.got
{'d'}
sage: D.get('not_here')
sage: sorted(list(D.got))
['d', 'not_here']
```

start()

We track which variables have been set or retrieved. This function initializes these lists to be empty.

EXAMPLES:

```
sage: from sage.doctest.util import RecordingDict
sage: D = RecordingDict(d = 42)
sage: D.set
set()
sage: D['a'] = 4
sage: D.set
{'a'}
sage: D.start(); D.set
set()
```

class sage.doctest.util.**Timer**

Bases: object

A simple timer.

EXAMPLES:

```
sage: from sage.doctest.util import Timer
sage: Timer()
{}
sage: TestSuite(Timer()).run()
```

annotate(*object*)

Annotates the given object with the cputime and walltime stored in this timer.

EXAMPLES:

```
sage: from sage.doctest.util import Timer
sage: Timer().start().annotate(EllipticCurve)
sage: EllipticCurve.cputime # random
2.817255
sage: EllipticCurve.walltime # random
1332649288.410404
```

start()

Start the timer.

Can be called multiple times to reset the timer.

EXAMPLES:

```
sage: from sage.doctest.util import Timer
sage: Timer().start()
{'cputime': ..., 'walltime': ...}
```

stop()

Stops the timer, recording the time that has passed since it was started.

EXAMPLES:

```
sage: from sage.doctest.util import Timer
sage: import time
sage: timer = Timer().start()
sage: time.sleep(float(0.5))
sage: timer.stop()
{'cputime': ..., 'walltime': ...}
```

sage.doctest.util.**count_noun**(*number*, *noun*, *plural=None*, *pad_number=False*, *pad_noun=False*)

EXAMPLES:

```
sage: from sage.doctest.util import count_noun
sage: count_noun(1, "apple")
'1 apple'
sage: count_noun(1, "apple", pad_noun=True)
'1 apple '
sage: count_noun(1, "apple", pad_number=3)
'  1 apple'
sage: count_noun(2, "orange")
'2 oranges'
sage: count_noun(3, "peach", "peaches")
'3 peaches'
sage: count_noun(1, "peach", plural='peaches', pad_noun=True)
'1 peach  '
```

sage.doctest.util.**dict_difference**(*self*, *other*)

Return a dict with all key-value pairs occurring in `self` but not in `other`.

EXAMPLES:

```
sage: from sage.doctest.util import dict_difference
sage: d1 = {1: 'a', 2: 'b', 3: 'c'}
sage: d2 = {1: 'a', 2: 'x', 4: 'c'}
sage: dict_difference(d2, d1)
{2: 'x', 4: 'c'}
```

```
sage: from sage.doctest.control import DocTestDefaults
sage: D1 = DocTestDefaults()
sage: D2 = DocTestDefaults(foobar='hello', timeout=100)
sage: dict_difference(D2.__dict__, D1.__dict__)
{'foobar': 'hello', 'timeout': 100}
```

sage.doctest.util.**make_recording_dict**(*D*, *st*, *gt*)

Auxiliary function for pickling.

EXAMPLES:

```
sage: from sage.doctest.util import make_recording_dict
sage: D = make_recording_dict({'a':4,'d':42},set([]),set(['not_here']))
sage: sorted(D.items())
[('a', 4), ('d', 42)]
sage: D.got
{'not_here'}
```

CHAPTER

NINE

FIXTURES TO HELP TESTING FUNCTIONALITY

Utilities which modify or replace code to help with doctesting functionality. Wrappers, proxies and mockups are typical examples of fixtures.

AUTHORS:

- Martin von Gagern (2014-12-15): AttributeAccessTracerProxy and trace_method
- Martin von Gagern (2015-01-02): Factor out TracerHelper and reproducible_repr

EXAMPLES:

You can use *trace_method()* to see how a method communicates with its surroundings:

```
sage: class Foo():
....:     def f(self):
....:         self.y = self.g(self.x)
....:     def g(self, arg):
....:         return arg + 1
....:
sage: foo = Foo()
sage: foo.x = 3
sage: from sage.doctest.fixtures import trace_method
sage: trace_method(foo, "f")
sage: foo.f()
enter f()
  read x = 3
  call g(3) -> 4
  write y = 4
exit f -> None
```

class sage.doctest.fixtures.**AttributeAccessTracerHelper**(*delegate*, *prefix=' '*, *reads=True*)

Bases: object

Helper to print proxied access to attributes.

This class does the actual printing of access traces for objects proxied by *AttributeAccessTracerProxy*. The fact that it's not a proxy at the same time helps avoiding complicated attribute access syntax.

INPUT:

- delegate – the actual object to be proxied
- prefix – (default: " ") string to prepend to each printed output
- reads – (default: True) whether to trace read access as well

EXAMPLES:

```
sage: class Foo():
....:     def f(self, *args):
....:         return self.x*self.x
....:
sage: foo = Foo()
sage: from sage.doctest.fixtures import AttributeAccessTracerHelper
sage: pat = AttributeAccessTracerHelper(foo)
sage: pat.set("x", 2)
  write x = 2
sage: pat.get("x")
  read x = 2
2
sage: pat.get("f")(3)
  call f(3) -> 4
4
```

get (*name*)

Read an attribute from the wrapped delegate object.

If that value is a method (i.e. a callable object which is not contained in the dictionary of the object itself but instead inherited from some class) then it is replaced by a wrapper function to report arguments and return value. Otherwise an attribute read access is reported.

EXAMPLES:

```
sage: class Foo():
....:     def f(self, *args):
....:         return self.x*self.x
....:
sage: foo = Foo()
sage: foo.x = 2
sage: from sage.doctest.fixtures import AttributeAccessTracerHelper
sage: pat = AttributeAccessTracerHelper(foo)
sage: pat.get("x")
  read x = 2
2
sage: pat.get("f")(3)
  call f(3) -> 4
4
```

set (*name*, *val*)

Write an attribute to the wrapped delegate object.

The name and new value are also reported in the output.

EXAMPLES:

```
sage: class Foo():
....:     pass
....:
sage: foo = Foo()
sage: from sage.doctest.fixtures import AttributeAccessTracerHelper
sage: pat = AttributeAccessTracerHelper(foo)
sage: pat.set("x", 2)
  write x = 2
sage: foo.x
2
```

class sage.doctest.fixtures.**AttributeAccessTracerProxy**(*delegate*, ***kwds*)

Bases: object

Proxy object which prints all attribute and method access to an object.

The implementation is kept lean since all access to attributes of the proxy itself requires complicated syntax. For this reason, the actual handling of attribute access is delegated to a *AttributeAccessTracerHelper*.

INPUT:

- delegate – the actual object to be proxied
- prefix – (default: " ") string to prepend to each printed output
- reads – (default: True) whether to trace read access as well

EXAMPLES:

```
sage: class Foo():
....:     def f(self, *args):
....:         return self.x*self.x
....:
sage: foo = Foo()
sage: from sage.doctest.fixtures import AttributeAccessTracerProxy
sage: pat = AttributeAccessTracerProxy(foo)
sage: pat.x = 2
  write x = 2
sage: pat.x
  read x = 2
2
sage: pat.f(3)
  call f(3) -> 4
4
```

__getattribute__(*name*)

Read an attribute from the wrapped delegate object.

If that value is a method (i.e. a callable object which is not contained in the dictionary of the object itself but instead inherited from some class) then it is replaced by a wrapper function to report arguments and return value. Otherwise an attribute read access is reported.

EXAMPLES:

```
sage: class Foo():
....:     def f(self, *args):
....:         return self.x*self.x
....:
sage: foo = Foo()
sage: foo.x = 2
sage: from sage.doctest.fixtures import AttributeAccessTracerProxy
sage: pat = AttributeAccessTracerProxy(foo)
sage: pat.x
  read x = 2
2
sage: pat.f(3)
  call f(3) -> 4
4
```

__setattr__(*name*, *val*)

Write an attribute to the wrapped delegate object.

The name and new value are also reported in the output.

EXAMPLES:

```
sage: class Foo():
....:     pass
....:
sage: foo = Foo()
sage: from sage.doctest.fixtures import AttributeAccessTracerProxy
sage: pat = AttributeAccessTracerProxy(foo)
sage: pat.x = 2
  write x = 2
sage: foo.x
2
```

sage.doctest.fixtures.**reproducible_repr**(*val*)

String representation of an object in a reproducible way.

This tries to ensure that the returned string does not depend on factors outside the control of the doctest. One example is the order of elements in a hash-based structure. For most objects, this is simply the `repr` of the object.

All types for which special handling had been implemented are covered by the examples below. If a doctest requires special handling for additional types, this function may be extended appropriately. It is an error if an argument to this function has a non-reproducible `repr` implementation and is not explicitly mentioned in an example case below.

INPUT:

- `val` – an object to be represented

OUTPUT:

A string representation of that object, similar to what `repr` returns but for certain cases with more guarantees to ensure exactly the same result for semantically equivalent objects.

EXAMPLES:

```
sage: from sage.doctest.fixtures import reproducible_repr
sage: print(reproducible_repr(set(["a", "c", "b", "d"])))
set(['a', 'b', 'c', 'd'])
sage: print(reproducible_repr(frozenset(["a", "c", "b", "d"])))
frozenset(['a', 'b', 'c', 'd'])
sage: print(reproducible_repr([1, frozenset("cab"), set("bar"), 0]))
[1, frozenset(['a', 'b', 'c']), set(['a', 'b', 'r']), 0]
sage: print(reproducible_repr({3.0: "three", "2": "two", 1: "one"}))            #␣
↪optional - sage.rings.real_mpfr
{'2': 'two', 1: 'one', 3.00000000000000: 'three'}
sage: print(reproducible_repr("foo\nbar"))  # demonstrate default case
'foo\nbar'
```

sage.doctest.fixtures.**trace_method**(*obj*, *meth*, ***kwds*)

Trace the doings of a given method. It prints method entry with arguments, access to members and other methods during method execution as well as method exit with return value.

INPUT:

- `obj` – the object containing the method
- `meth` – the name of the method to be traced
- `prefix` – (default: `"  "`) string to prepend to each printed output

- `reads` – (default: `True`) whether to trace read access as well

EXAMPLES:

```
sage: class Foo():
....:     def f(self, arg=None):
....:         self.y = self.g(self.x)
....:         if arg: return arg*arg
....:     def g(self, arg):
....:         return arg + 1
....:
sage: foo = Foo()
sage: foo.x = 3
sage: from sage.doctest.fixtures import trace_method
sage: trace_method(foo, "f")
sage: foo.f()
enter f()
  read x = 3
  call g(3) -> 4
  write y = 4
exit f -> None
sage: foo.f(3)
enter f(3)
  read x = 3
  call g(3) -> 4
  write y = 4
exit f -> 9
9
```

CHAPTER

TEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

d

INDEX

Non-alphabetical

A

B

C

D

E

F

R

S

T

U

W