Sage Reference Manual: Sage's Doctesting Framework

Release 6.10

The Sage Development Team

December 21, 2015

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: `sage.structure.sage_object.SageObject`

This class controls doctesting of files.

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

`add_files`()

Checks for the flags '–all', '–new' and '–sagenb'.

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 os
sage: log_location = os.path.join(SAGE_TMP, 'control_dt_log.log')
sage: DD = DocTestDefaults(all=True, logfile=log_location)
sage: DC = DocTestController(DD, [])
sage: DC.add_files()
Doctesting entire Sage library.
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 ...

sage: DD = DocTestDefaults(sagenb = True)
sage: DC = DocTestController(DD, [])
sage: DC.add_files()
Doctesting the Sage notebook.
sage: DC.files[0][-6:]
'sagenb'
```

`cleanup`(*final=True*)

Runs 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', 'infinity.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': 0.1*(i+1)}
....:

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

create_run_id()

Creates 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()

Expands `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)
10
```

```
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: sorted(list(DC.sources[0].options.optional))
['guava', 'magma']
```

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': 0.1*(i+1)}
sage: DC.stats['sage.doctest.control'] = {'failed':True,'walltime':1.0}
sage: DC.filter_sources()
Only doctesting files that failed last test.
sage: len(DC.sources)
1
```

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':{u'walltime':1.0r}}, stats_file)
```

```
sage: DC.load_stats(filename)
sage: DC.stats['sage.doctest.control']
{u'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']
{u'walltime': 1.0}
```

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

Logs the string `s + end` (where `end` is a newline by default) to the logfile and prints 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: print open(DD.logfile).read()
hello world
```

Check that no duplicate logs appear, even when forking (trac ticket #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: print open(DD.logfile).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()
Running doctests with ID ...
Doctesting 1 file.
sage -t .../sage/sets/non_negative_integers.py
    [... tests, ... s]
----------------------------------------------------------------------
All tests passed!
```

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

run_doctests()

Actually runs 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()
sage: DC.run_doctests()
Doctesting 1 file.
sage -t .../sage/rings/homset.py
    [... tests, ... s]
------------------------------------------------------------------------
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 or valgrind.

INPUT:

- `testing` – boolean; 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 -x "$SAGE_LOCAL/bin/sage-gdb-commands" --args python "$SAGE_LOCAL/bin/sage-runtests

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 --su
```

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'] = {u'walltime':1.0r}
sage: filename = tmp_filename()
sage: DC.save_stats(filename)
sage: import json
sage: D = json.load(open(filename))
sage: D['sage.doctest.control']
{u'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. 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': 0.1*(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.reporting
sage.doctest.parsing
sage.doctest.forker
sage.doctest.fixtures
sage.doctest.control
sage.doctest.all
sage.doctest
```

test_safe_directory(*dir=None*)

Test that the given directory is safe to run Python code from.

We use the check added to Python for this, which gives a warning when the current directory is considered unsafe. We promote this warning to an error with `-Werror`. See `sage/tests/cmdline.py` for a doctest that this works, see also trac ticket #13579.

TESTS:

```
sage: from sage.doctest.control import DocTestDefaults, DocTestController
sage: DD = DocTestDefaults()
sage: DC = DocTestController(DD, [])
sage: DC.test_safe_directory()
sage: d = os.path.join(tmp_dir(), "test")
sage: os.mkdir(d)
sage: os.chmod(d, 0o777)
sage: DC.test_safe_directory(d)
Traceback (most recent call last):
...
RuntimeError: refusing to run doctests...
```

class `sage.doctest.control.`**`DocTestDefaults`**`(`*****kwds*** `)`

Bases: `sage.structure.sage_object.SageObject`

This class is used for doctesting the Sage doctest module.

It fills in attributes to be the same as the defaults defined in `SAGE_LOCAL/bin/sage-runtests`, expect for a few places, which is mostly to make doctesting more predictable.

EXAMPLES:

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

Keyword arguments become attributes:

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

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

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

`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*)

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

EXAMPLES:

```
sage: from sage.doctest.control import skipfile
sage: skipfile("skipme.c")
True
sage: f = tmp_filename(ext=".pyx")
sage: skipfile(f)
False
sage: open(f, "w").write("# nodoctest")
sage: skipfile(f)
True
```

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.

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

Bases: `sage.doctest.sources.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: from sage.env import SAGE_SRC
sage: import os
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','sources.py')
sage: FDS = FileDocTestSource(filename,DocTestDefaults())
sage: FDS.basename
'sage.doctest.sources'
```

TESTS:

```
sage: TestSuite(FDS).run()
```

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: from sage.env import SAGE_SRC
sage: import os
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*)

Returns a list of doctests for this file.

INPUT:

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

OUTPUT:

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

EXAMPLES:

```
sage: from sage.doctest.control import DocTestDefaults
sage: from sage.doctest.sources import FileDocTestSource
sage: from sage.env import SAGE_SRC
sage: import os
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','sources.py')
sage: FDS = FileDocTestSource(filename,DocTestDefaults())
sage: doctests, extras = FDS.create_doctests(globals())
sage: len(doctests)
40
sage: extras['tab']
False
```

We give a self referential example:

```
sage: doctests[17].name
'sage.doctest.sources.FileDocTestSource.create_doctests'
sage: doctests[17].examples[10].source
'doctests[Integer(17)].examples[Integer(10)].source\n'
```

TESTS:

We check that we correctly process results that depend on 32 vs 64 bit architecture:

```
sage: import sys
sage: bitness = '64' if sys.maxsize > (1 << 32) else '32'
sage: n = -920390823904823094890238490238484; hash(n) > 0
False # 32-bit
True  # 64-bit
sage: ex = doctests[17].examples[13]
sage: (bitness == '64' and ex.want == 'True  \n') or (bitness == '32' and ex.want == 'False
True
```

We check that lines starting with a # aren't doctested:

```
#sage: raise RuntimeError
```

in_lib()

Whether this file should be considered part of the Sage library.

Such files aren't loaded before running tests.

EXAMPLES:

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

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

```
sage: import os
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','sources.py')
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` – a 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: from sage.env import SAGE_SRC
sage: import os
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','sources.py')
sage: FDS = FileDocTestSource(filename,DocTestDefaults())
sage: FDS._init()
sage: FDS.quotetype = "'''"
sage: FDS.ending_docstring("'''")
<_sre.SRE_Match object at ...>
sage: FDS.ending_docstring('\"\"\"')
```

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` – a 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: from sage.env import SAGE_SRC
sage: import os
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','sources.py')
sage: FDS = FileDocTestSource(filename,DocTestDefaults())
sage: FDS._init()
sage: FDS.starting_docstring("r'''")
<_sre.SRE_Match object at ...>
sage: FDS.ending_docstring("'''")
<_sre.SRE_Match object at ...>
sage: FDS.qualified_name = NestedName(FDS.basename)
```

```
sage: FDS.starting_docstring("class MyClass(object):")
sage: FDS.starting_docstring("    def hello_world(self):")
sage: FDS.starting_docstring("        '''")
<_sre.SRE_Match object at ...>
sage: FDS.qualified_name
sage.doctest.sources.MyClass.hello_world
sage: FDS.ending_docstring("    '''")
<_sre.SRE_Match object at ...>
sage: FDS.starting_docstring("class NewClass(object):")
sage: FDS.starting_docstring("    '''")
<_sre.SRE_Match object at ...>
sage: FDS.qualified_name
sage.doctest.sources.NewClass
```

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

Bases: `sage.doctest.sources.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` – a string, one line of an input file

OUTPUT:

- a 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(False, 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)
```

starting_docstring(*line*)

A line ending with a double quote starts a verbatim block in a ReST file.

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

INPUT:

- `line` – a 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`**

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` – a string containing documentation and tests.
- `namespace` – a dictionary or `sage.doctest.util.RecordingDict`.
- `start` – an 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: from sage.env import SAGE_SRC
sage: import os
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','util.py')
sage: FDS = FileDocTestSource(filename,DocTestDefaults())
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: `sage.doctest.sources.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` – a string, giving the source code to be parsed for doctests.
- `options` – a `sage.doctest.control.DocTestDefaults` or equivalent.
- `printpath` – a string, to be used in place of a filename when doctest failures are displayed.
- `lineno_shift` – an 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
sage: PSS = PythonStringSource('<runtime>', s, DocTestDefaults(), 'runtime')
sage: dt, extras = PSS.create_doctests({})
sage: extras['line_number']
True
```

create_doctests(*namespace*)

Creates doctests from this string.

INPUT:

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

OUTPUT:

- `doctests` – a 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, PythonSo
sage: PSS = PythonStringSource('<runtime>', s, DocTestDefaults(), 'runtime')
sage: dt, tabs = PSS.create_doctests({})
sage: for t in dt:
....:     print t.name, t.examples[0].sage_source
<runtime> 2 + 2
```

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

Bases: `sage.doctest.sources.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` – a string, one line of an input file
- `check_skip` – boolean (default True), used internally in starting_docstring.

OUTPUT:

- a boolean giving 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}")
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
```

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` – a string, one line of an input file

OUTPUT:

- a boolean giving 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
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: from sage.env import SAGE_SRC
sage: import os
```

```
sage: get_basename(os.path.join(SAGE_SRC,'sage','doctest','sources.py'))
'sage.doctest.sources'
```

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.

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

Bases: `sage.structure.sage_object.SageObject`

Creates 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: from sage.env import SAGE_SRC
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
    [... tests, ... s]
sage -t .../sage/rings/big_oh.py
    [... tests, ... s]
```

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: from sage.env import SAGE_SRC
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]
sage -t .../rings/big_oh.py
    [... tests, ... s]
```

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: from sage.env import SAGE_SRC
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]
sage -t .../rings/ideal.py
    [... tests, ... s]
```

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: from sage.env import SAGE_SRC
sage: import os
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','sources.py')
sage: DD = DocTestDefaults()
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', 'walltime']
```

class `sage.doctest.forker.`**`DocTestWorker`**(*source*, *options*, *funclist=*[ ])

Bases: `multiprocessing.process.Process`

The DocTestWorker process runs one `DocTestTask` for a given source. It returns messages about doctest failures (or all tests if verbose doctesting) though 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` – a list of callables to be called at the start of the child process.

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: from sage.env import SAGE_SRC
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','util.py')
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]
```

kill()

Kill this worker. The first time this is called, use `SIGHUP`. 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: from sage.env import SAGE_SRC
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 `SIGHUP`, such that killing will fail initially:

```
sage: from sage.ext.pselect import PSelecter
sage: import signal
sage: def block_hup():
....:     # We never __exit__()
....:     PSelecter([signal.SIGHUP]).__enter__()
sage: W = DocTestWorker(FDS, DD, [block_hup])
sage: W.start()
sage: W.killed
False
sage: W.kill()
sage: W.killed
True
sage: time.sleep(0.2)   # Worker doesn't die
sage: W.kill()          # Worker dies now
sage: time.sleep(0.2)
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: from sage.env import SAGE_SRC
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','util.py')
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()
```

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

run()

Runs the DocTestTask under its own PGID.

TESTS:

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

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: from sage.env import SAGE_SRC
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','util.py')
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', 'walltime']
sage: len(W.output) > 0
True
```

start()

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

TESTS:

```
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: from sage.env import SAGE_SRC
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','util.py')
sage: DD = DocTestDefaults()
sage: FDS = FileDocTestSource(filename,DD)
sage: W = DocTestWorker(FDS, DD)
sage: W.start()
```

```
sage: try:
....:     os.fstat(W.wmessages)
....: except OSError:
....:     print "Write end of pipe successfully closed"
Write end of pipe successfully closed
sage: W.join()  # Wait for worker to finish
```

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

Bases: `doctest.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.

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=d
sage: DTR
<sage.doctest.forker.SageDocTestRunner instance at ...>
```

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

Runs 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` – a 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: from sage.env import SAGE_SRC
sage: import doctest, sys, os, hashlib
sage: DTR = SageDocTestRunner(SageOutputChecker(), verbose=False, sage_options=DD, optionfla
sage: DTR.running_doctest_digest = hashlib.md5()
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','forker.py')
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: compiler = lambda ex: compile(ex.source, '<doctest sage.doctest.forker[0]>', 'single',
sage: DTR.compile_and_execute(ex0, compiler, globs)
1764
sage: globs['doctest_var']
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: compiler = lambda ex:compile(ex.source, '<doctest sage.doctest.forker[1]>', 'single',
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: compiler = lambda ex:compile(ex.source, '<doctest sage.doctest.forker[2]>', 'single',
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` – a string, the result of running `example`
- `globs` – a 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: from sage.env import SAGE_SRC
sage: import doctest, sys, os
sage: DTR = SageDocTestRunner(SageOutputChecker(), verbose=True, sage_options=DD, optionflag
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','forker.py')
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 11, 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: import os
sage: os.environ['SAGE_PEXPECT_LOG'] = "1"
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("sdf.init_sage()")
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_o
sage: sage0._prompt = r"debug: "
sage: print sage0.eval("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.eval("a")
'...17'
sage: sage0._prompt = "sage: "
sage: sage0.eval("quit")
'Returning to doctests...TestResults(failed=1, attempted=2)'
```

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

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` – a string, the result of running `example`

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.misc.misc import walltime
sage: from sage.env import SAGE_SRC
sage: import doctest, sys, os
sage: DTR = SageDocTestRunner(SageOutputChecker(), verbose=True, sage_options=DD, optionflag
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','forker.py')
sage: FDS = FileDocTestSource(filename,DD)
sage: doctests, extras = FDS.create_doctests(globals())
sage: ex = doctests[0].examples[0]
sage: ex.walltime = 1.23
sage: DTR.report_overtime(sys.stdout.write, doctests[0], ex, 'BAD ANSWER\n')
**********************************************************************
File ".../sage/doctest/forker.py", line 11, in sage.doctest.forker
Warning, slow doctest:
    doctest_var = 42; doctest_var^2
Test ran for 1.23 s
```

`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: from sage.env import SAGE_SRC
sage: import doctest, sys, os
sage: DTR = SageDocTestRunner(SageOutputChecker(), verbose=True, sage_options=DD, optionflag
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','forker.py')
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 11):    doctest_var = 42; doctest_var^2
Expecting:
    1764
```

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

Called when an example succeeds.

INPUT:

•`out` – a function for printing

•`test` – a `doctest.DocTest` instance

•`example` – a `doctest.Example` instance in `test`

•`got` – a string, the result of running `example`

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.misc.misc import walltime
sage: from sage.env import SAGE_SRC
sage: import doctest, sys, os
sage: DTR = SageDocTestRunner(SageOutputChecker(), verbose=True, sage_options=DD, optionflag
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','forker.py')
sage: FDS = FileDocTestSource(filename,DD)
sage: doctests, extras = FDS.create_doctests(globals())
sage: ex = doctests[0].examples[0]
sage: ex.walltime = 0.0
sage: DTR.report_success(sys.stdout.write, doctests[0], ex, '1764')
ok [0.00 s]
```

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: import os
sage: os.environ['SAGE_PEXPECT_LOG'] = "1"
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 Ellipt
sage: _ = sage0.eval("DT = doctest.DocTest([ex], globals(), 'singular_curve', None, 0, None)
sage: _ = sage0.eval("DTR = sdf.SageDocTestRunner(SageOutputChecker(), verbose=False, sage_o
sage: sage0._prompt = r"\(Pdb\) "
sage: sage0.eval("DTR.run(DT, clear_globs=False)") # indirect doctest
'... ArithmeticError("invariants " + str(ainvs) + " define a singular curve")'
sage: sage0.eval("l")
'...if self.discriminant() == 0:...raise ArithmeticError...'
sage: sage0.eval("u")
'...EllipticCurve_field.__init__(self, K, ainvs)'
sage: sage0.eval("p ainvs")
'(0, 0, 0, 0, 0)'
sage: sage0._prompt = "sage: "
sage: sage0.eval("quit")
'TestResults(failed=1, attempted=1)'
```

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

Runs 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` – the set of compiler flags used to execute examples (passed in to the `compile()`). If None, they are filled in from the result of `doctest._extract_future_flags()` applied to `test.globs`.
- `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: from sage.env import SAGE_SRC
sage: import doctest, sys, os
sage: DTR = SageDocTestRunner(SageOutputChecker(), verbose=False, sage_options=DD, optionfla
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','forker.py')
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, optionfla
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: from sage.env import SAGE_SRC
sage: import doctest, sys, os, hashlib
sage: DTR = SageDocTestRunner(SageOutputChecker(), verbose=False, sage_options=DD, optionfla
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','forker.py')
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` – a 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: from sage.env import SAGE_SRC
sage: import doctest, sys, os
sage: DTR = SageDocTestRunner(SageOutputChecker(), verbose=False, sage_options=DD, optionfla
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','forker.py')
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.iteritems()))
[('cputime', [...]), ('err', None), ('failures', 0), ('walltime', [...])]
```

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

Bases: `sage.structure.sage_object.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
```

getvalue()

Gets 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’
```

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
```

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

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

TESTS:

```
sage: import signal
sage: from sage.doctest.forker import dummy_handler
sage: _ = signal.signal(signal.SIGUSR1, dummy_handler)
sage: os.kill(os.getpid(), signal.SIGUSR1)
sage: signal.signal(signal.SIGUSR1, signal.SIG_DFL)
<function dummy_handler at ...>
```

sage.doctest.forker.**init_sage**()

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: 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

2
  + 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.`**`warning_function`**(*file*)

Creates a function that prints warnings to the given file.

INPUT:

- `file` – an open file handle.

OUTPUT:

- a function that prings warnings to the given file.

EXAMPLES:

```
sage: from sage.doctest.forker import warning_function
sage: import tempfile
sage: F = tempfile.TemporaryFile()
sage: wrn = warning_function(F)
sage: wrn("bad stuff", UserWarning, "myfile.py", 0)
sage: F.seek(0)
sage: F.read()
'doctest:...: UserWarning: bad stuff\n'
```

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 (trac ticket #16889).

class `sage.doctest.parsing.`**`MarkedOutput`**

Bases: `str`

A subclass of string with context for whether another string matches it.

EXAMPLES:

```
sage: from sage.doctest.parsing import MarkedOutput
sage: s = MarkedOutput("abc")
sage: s.rel_tol
0
sage: s.update(rel_tol = .05)
'abc'
sage: s.rel_tol
0.0500000000000000
```

`update``(`***kwds*`)`

EXAMPLES:

```
sage: from sage.doctest.parsing import MarkedOutput
sage: s = MarkedOutput("0.0007401")
sage: s.update(abs_tol = .0000001)
'0.0007401'
sage: s.rel_tol
0
sage: s.abs_tol
1.00000000000000e-7
```

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

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: from sage.env import SAGE_SRC
sage: import os
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','forker.py')
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`**(*long=False*, *optional_tags=()*)

Bases: `doctest.DocTestParser`

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

`parse`(*string*, **args*)

A Sage specialization of `doctest.DocTestParser`.

INPUT:

- `string` – the string to parse.
- `name` – optional string giving the name indentifying 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 or 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(True, ('sage','magma','guava'))
sage: example = 'Explanatory text::\n\n    sage: E = magma("EllipticCurve([1, 1, 1, -10, -10
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(True, ('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.parsing.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, but works in Sage 5.5:

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

It also works without the line continuation:

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

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

Bases: doctest.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(True, ('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.parsing.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
```

add_tolerance (*wantval*, *want*)

Enlarge the real interval element `wantval` according to the tolerance options in `want`.

INPUT:

- `wantval` – a real interval element
- `want` – a `MarkedOutput` describing the tolerance

OUTPUT:

- an interval element containing `wantval`

EXAMPLES:

```
sage: from sage.doctest.parsing import MarkedOutput, SageOutputChecker
sage: OC = SageOutputChecker()
sage: want_tol = MarkedOutput().update(tol=0.0001)
sage: want_abs = MarkedOutput().update(abs_tol=0.0001)
sage: want_rel = MarkedOutput().update(rel_tol=0.0001)
sage: OC.add_tolerance(pi.n(64), want_tol).endpoints()
(3.14127849432443, 3.14190681285516)
sage: OC.add_tolerance(pi.n(64), want_abs).endpoints()
(3.14149265358979, 3.14169265358980)
sage: OC.add_tolerance(pi.n(64), want_rel).endpoints()
(3.14127849432443, 3.14190681285516)
sage: OC.add_tolerance(1e1000, want_tol)
1.000?e1000
sage: OC.add_tolerance(1e1000, want_abs)
1.000000000000000?e1000
sage: OC.add_tolerance(1e1000, want_rel)
1.000?e1000
sage: OC.add_tolerance(0, want_tol)
0.000?
sage: OC.add_tolerance(0, want_abs)
0.000?
sage: OC.add_tolerance(0, want_rel)
0
```

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

Checks to see if the output matches the desired output.

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

INPUT:

- `want` – a string or `MarkedOutput`
- `got` – a string
- `optionflags` – an 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"
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
sage: raise RuntimeError   # rel tol 1e10
Traceback (most recent call last):
```

```
...
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
```

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

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

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
```

TESTS:

```
sage: from sage.doctest.parsing import SageOutputChecker
sage: OC = SageOutputChecker()
sage: teststr = '-'.join([
....:     'bold\x1b[1m',
....:     'red\x1b[31m',
....:     'oscmd\x1ba'])
sage: OC.human_readable_escape_sequences(teststr)
'bold<CSI-1m>-red<CSI-31m>-oscmd<ESC-a>'
```

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` – a string
- `optionflags` – an integer, passed down to `doctest.OutputChecker`

OUTPUT:

- a 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").upda
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-01 > 1e-01

sage: print OC.output_difference(tentol,zero,optflag)
Expected:
    10.0
Got:
    0.0
Tolerance exceeded:
    10.0 vs 0.0, tolerance 1e+00 > 1e-01

sage: print OC.output_difference(tentol,eps,optflag)
Expected:
    10.0
Got:
    -0.05
Tolerance exceeded:
    10.0 vs -0.05, tolerance 1e+00 > 1e-01

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 1e+00 > 1e+00
    10.0 vs 11.2, tolerance 1e+00 > 1e+00
```

TESTS:

```
sage: print OC.output_difference(tenabs,zero,optflag)
Expected:
    10.0
Got:
    0.0
Tolerance exceeded:
    10.0 vs 0.0, tolerance 1e+01 > 1e-01

sage: print OC.output_difference(tenrel,zero,optflag)
Expected:
    10.0
Got:
    0.0
Tolerance exceeded:
    10.0 vs 0.0, tolerance 1e+00 > 1e-01
```

```
sage: print OC.output_difference(tenrel,eps,optflag)
Expected:
    10.0
Got:
    -0.05
Tolerance exceeded:
    10.0 vs -0.05, tolerance 1e+00 > 1e-01

sage: print OC.output_difference(zerotol,ten,optflag)
Expected:
    0.0
Got:
    10.05
Tolerance exceeded:
    0.0 vs 10.05, tolerance 1e+01 > 1e-01

sage: print OC.output_difference(zeroabs,ten,optflag)
Expected:
    0.0
Got:
    10.05
Tolerance exceeded:
    0.0 vs 10.05, tolerance 1e+01 > 1e-01

sage: print OC.output_difference(zerorel,eps,optflag)
Expected:
    0.0
Got:
    -0.05
Tolerance exceeded:
    0.0 vs -0.05, tolerance inf > 1e-01

sage: print OC.output_difference(zerorel,ten,optflag)
Expected:
    0.0
Got:
    10.05
Tolerance exceeded:
    0.0 vs 10.05, tolerance inf > 1e-01
```

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

Returns 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.**make_marked_output**(*s*, *D*)

Auxilliary function for pickling.

EXAMPLES:

```
sage: from sage.doctest.parsing import make_marked_output
sage: s = make_marked_output("0.0007401",{'abs_tol':.0000001})
sage: s
'0.0007401'
sage: s.abs_tol
1.00000000000000e-7
```

`sage.doctest.parsing.`**`parse_optional_tags`**`(`*string*`)`

Returns a set consisting of the 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'
- 'optional: PKG_NAME' – the set will just contain 'PKG_NAME'

EXAMPLES:

```
sage: from sage.doctest.parsing import parse_optional_tags
sage: parse_optional_tags("sage: magma('2 + 2')# optional: magma")
{'magma'}
sage: parse_optional_tags("sage: #optional -- mypkg")
{'mypkg'}
sage: parse_optional_tags("sage: print(1)  # parentheses are optional here")
set()
sage: parse_optional_tags("sage: print(1)  # optional")
{''}
sage: sorted(list(parse_optional_tags("sage: #optional -- foo bar, baz")))
['bar', 'foo']
sage: sorted(list(parse_optional_tags("    sage: factor(10^(10^10) + 1) # LoNg TiME, NoT TeSTED;
['long time', 'not tested', 'p4cka9e']
sage: parse_optional_tags("    sage: raise RuntimeError # known bug")
{'bug'}
sage: sorted(list(parse_optional_tags("    sage: determine_meaning_of_life() # long time, not im
['long time', 'not implemented']
```

We don't parse inside strings:

```
sage: parse_optional_tags("    sage: print '  # long time'")
set()
sage: parse_optional_tags("    sage: print '  # long time'  # not tested")
{'not tested'}
```

UTF-8 works:

```
sage: parse_optional_tags("'ěščřžýáíéd'Ď'")
set()
```

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

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

INPUT:

- `source` – a string, the source of a doctest

- `want` – a 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)
<type '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*)

Returns a symmetric function of the arguments as hex strings.

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

EXAMPLES:

```
sage: from sage.doctest.parsing import reduce_hex
sage: reduce_hex(["abc", "12399aedf"])
'0000000000000000000000012399a463'
sage: reduce_hex(["12399aedf","abc"])
'0000000000000000000000012399a463'
```

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 :class: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 non-zero 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: `sage.structure.sage_object.SageObject`

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

`finalize``()`

Print out the postcript 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: from sage.env import SAGE_SRC
sage: import os, sys, doctest
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','reporting.py')
sage: DD = DocTestDefaults()
sage: FDS = FileDocTestSource(filename,DD)
```

```
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,optionf
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]
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 outp
    [... tests, 1 failure, ... s]
```

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()

----------------------------------------------------------------------
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` – a boolean, whether doctests timed out
- `return_code` – an int, 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` – a string, printed if there was some kind of failure
- `pid` – optional 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: from sage.env import SAGE_SRC
sage: import os, sys, doctest
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','reporting.py')
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, '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, 'walltime': 1000000.0}}
```

Or a process that segfaulted:

```
sage: import signal
sage: DTR.report(FDS, False, -signal.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, 'walltime': 1000000.0}}
```

Report a timeout with results and a `SIGKILL`:

```
sage: DTR.report(FDS, True, -signal.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, '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, '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, optior
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]
sage: DTR.stats
{'sage.doctest.reporting': {'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 outp
    [... tests, 1 failure, ... s]
```

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

```
sage: DC.options = DocTestDefaults(show_skipped=True)
sage: import collections
sage: optionals = collections.defaultdict(int)
sage: optionals['magma'] = 5; optionals['long time'] = 4; optionals[''] = 1; optionals['not
```

```
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 other tests skipped
    [... tests, ... s]
```

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'
```

report_head(*source*)

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

INPUT:

- `source` – a source from `sage.doctest.sources`

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: from sage.env import SAGE_SRC
sage: filename = os.path.join(SAGE_SRC,'sage','doctest','reporting.py')
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.doctest.reporting.**signal_name**(*sig*)

Return a string describing a signal number.

EXAMPLES:

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

CHAPTER

SIX

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 trac ticket #14370:

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

Check that trac ticket #2235 has been fixed:

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

Check handling of tolerances:

```
sage: subprocess.call(["sage", "-t", "--warn-long", "0", "tolerance.rst"], **kwds)  # long time
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 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 1e-06 > 1e-06
**********************************************************************
...
1
```

Test the `--initial` option:

```
sage: subprocess.call(["sage", "-t", "--warn-long", "0", "-i", "initial.rst"], **kwds)   # long time
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 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 initial.rst  # 5 doctests failed
----------------------------------------------------------------------
...
1
```

Test a timeout using the `SAGE_TIMEOUT` environment variable:

```
sage: from copy import deepcopy
sage: kwds2 = deepcopy(kwds)
sage: kwds2['env']['SAGE_TIMEOUT'] = "3"
sage: subprocess.call(["sage", "-t",  "--warn-long", "0", "99seconds.rst"], **kwds2)  # long time
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 99seconds.rst
    Timed out
**********************************************************************
Tests run before process (pid=...) timed out:
...
----------------------------------------------------------------------
sage -t --warn-long 0.0 99seconds.rst  # Timed out
----------------------------------------------------------------------
...
4
```

Test handling of `KeyboardInterrupt` in doctests:

```
sage: subprocess.call(["sage", "-t",  "--warn-long", "0", "keyboardinterrupt.rst"], **kwds)  # long
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 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 keyboardinterrupt.rst  # 1 doctest failed
----------------------------------------------------------------------
...
1
```

Interrupt the doctester:

```
sage: subprocess.call(["sage", "-t",  "--warn-long", "0", "interrupt.rst"], **kwds)  # long time
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 interrupt.rst
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
....:      "--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 in max(20, 120 * 0.05) = 20 seconds:

```
sage: pid = int(open(F).read())     # long time
sage: time.sleep(2)                 # long time
sage: os.kill(pid, signal.SIGHUP)   # long time; 2 seconds passed => still alive
sage: time.sleep(23)                # long time
sage: os.kill(pid, signal.SIGHUP)   # long time; 25 seconds passed => dead
Traceback (most recent call last):
...
OSError: ...
```

Test a doctest failing with `abort()`:

```
sage: subprocess.call(["sage", "-t",  "--warn-long", "0", "abort.rst"], **kwds)  # long time
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 abort.rst
    Killed due to abort
**********************************************************************
Tests run before process (pid=...) failed:
...
------------------------------------------------------------------------
Unhandled SIGABRT: An abort() occurred in Sage.
This probably occurred because a *compiled* component of Sage has a bug
in it and is not properly wrapped with sig_on(), sig_off().
Sage will now terminate.
------------------------------------------------------------------------
...
----------------------------------------------------------------------
sage -t --warn-long 0.0 abort.rst  # Killed due to abort
----------------------------------------------------------------------
...
16
```

A different kind of crash:

```
sage: subprocess.call(["sage", "-t",  "--warn-long", "0", "fail_and_die.rst"], **kwds)  # long time
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 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 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", "sig_on.rst"], **kwds)  # long time
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 sig_on.rst
**********************************************************************
File "sig_on.rst", line 5, in sage.doctest.tests.sig_on
Failed example:
    sig_on_count() # check sig_on/off pairings (virtual doctest)
Expected:
    0
Got:
    1
**********************************************************************
1 item had failures:
   1 of   4 in sage.doctest.tests.sig_on
    [2 tests, 1 failure, ...]
----------------------------------------------------------------------
sage -t --warn-long 0.0 sig_on.rst  # 1 doctest failed
----------------------------------------------------------------------
...
1
```

Test the `--debug` option:

```
sage: subprocess.call(["sage", "-t",  "--warn-long", "0", "--debug", "simple_failure.rst"], stdin=ope
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 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
debug:

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 simple_failure.rst  # 1 doctest failed
----------------------------------------------------------------------
...
1
```

Test running under gdb, without and with a timeout:

```
sage: subprocess.call(["sage", "-t",  "--warn-long", "0", "--gdb", "1second.rst"], stdin=open(os.devn
exec gdb ...
Running doctests...
Doctesting 1 file.
sage -t... 1second.rst
    [2 tests, ... s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
...
0
```

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

```
sage: subprocess.call(["sage", "-t", "--gdb",  "--warn-long", "0", "-T30", "99seconds.rst"], stdin=op
exec gdb ...
Running doctests...
    Timed out
4
```

Test the `--show-skipped` option:

```
sage: subprocess.call(["sage", "-t",  "--warn-long", "0", "--show-skipped", "show_skipped.rst"], **kw
Running doctests ...
Doctesting 1 file.
sage -t --warn-long 0.0 show_skipped.rst
    1 unlabeled test not run
    2 tests not run due to known bugs
    1 gap test not run
    1 long test not run
    1 other test skipped
    [1 test, ... s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
...
0
```

Optional tests are run correctly:

```
sage: subprocess.call(["sage", "-t",  "--warn-long", "0", "--long", "--show-skipped", "--optional=sag
Running doctests ...
Doctesting 1 file.
sage -t --long --warn-long 0.0 show_skipped.rst
    1 unlabeled test not run
    2 tests not run due to known bugs
    1 other test skipped
    [3 tests, ... s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
...
0

sage: subprocess.call(["sage", "-t",  "--warn-long", "0", "--long", "--show-skipped", "--optional=gAp
Running doctests ...
Doctesting 1 file.
sage -t --long --warn-long 0.0 show_skipped.rst
    1 unlabeled test not run
    2 tests not run due to known bugs
    1 sage test not run
    1 other test skipped
    [2 tests, ... s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
...
0
```

Test an invalid value for `--optional`:

```
sage: subprocess.call(["sage", "-t",  "--warn-long", "0", "--optional=bad-option", "show_skipped.rst
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", "atexit.rst"], **kwds2)  # long time
Running doctests...
Doctesting 1 file.
sage -t --warn-long 0.0 atexit.rst
    [3 tests, ... s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
...
0
sage: os.path.isfile(F)  # long time
False
sage: try:
....:     os.unlink(F)
....: except OSError:
....:     pass
```

CHAPTER

SEVEN

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*`)`

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
```

TESTS:

```
sage: TestSuite(qname).run()
```

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
```

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

TESTS:

```
sage: TestSuite(D).run()
```

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**

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(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 occuring 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*)

Auxilliary 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 EIGHT

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(object):
....:     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.

EXAMPLE:

```
sage: class Foo(object):
....:     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.

EXAMPLE:

```
sage: class Foo(object):
....:     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.

EXAMPLE:

```
sage: class Foo(object):
....:     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.

EXAMPLE:

```
sage: class Foo(object):
....:     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.

EXAMPLE:

```
sage: class Foo(object):
....:     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.

EXAMPLE:

```
sage: class Foo(object):
....:     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 apropriately. It is an error if an argument to this function has a non-reproducible `repr` implementation and is not explicitely 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.

EXAMPLE:

```
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"}))
{'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.

EXAMPLE:

```
sage: class Foo(object):
....:     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

NINE

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX