nose Documentation

Release 1.3.7

Jason Pellerin

Aug 30, 2017

Contents

Nose has been in maintenance mode for the past several years and will likely cease without a new person/team to take over maintainership. New projects should consider using Nose2, py.test, or just plain unittest/unittest2.

CHAPTER 1

Installation and quick start

On most UNIX-like systems, you'll probably need to run these commands as root or using sudo.

Install nose using setuptools/distribute:

```
easy_install nose
```

Or pip:

```
pip install nose
```

Or, if you don't have setuptools/distribute installed, use the download link at right to download the source package, and install it in the normal fashion: Ungzip and untar the source package, cd to the new directory, and:

```
python setup.py install
```

However, **please note** that without setuptools/distribute installed, you will not be able to use third-party nose plugins.

This will install the nose libraries, as well as the *nosetests* script, which you can use to automatically discover and run tests.

Now you can run tests for your project:

```
cd path/to/project
nosetests
```

You should see output something like this:

```
.................................
----------------------------------------------------------------------
Ran 34 tests in 1.440s

OK
```

Indicating that nose found and ran your tests.

For help with nosetests' many command-line options, try:

```
nosetests -h
```

or visit the *usage documentation*.

CHAPTER 2

Python3

nose supports python3. Building from source on python3 requires distribute. If you don't have distribute installed, `python3 setup.py install` will install it via distribute's bootstrap script.

Additionally, if your project is using 2to3, `python3 setup.py nosetests` command will automatically convert your sources with 2to3 and then run the tests with python 3.

Warning: nose itself supports python 3, but many 3rd-party plugins do not!

Testing with nose

Writing tests is easier

nose collects tests from `unittest.TestCase` subclasses, of course. But you can also write simple test functions, as well as test classes that are *not* subclasses of `unittest.TestCase`. nose also supplies a number of helpful functions for writing timed tests, testing for exceptions, and other common use cases. See *Writing tests* and *Testing tools* for more.

Running tests is easier

nose collects tests automatically, as long as you follow some simple guidelines for organizing your library and test code. There's no need to manually collect test cases into test suites. Running tests is responsive, since nose begins running tests as soon as the first test module is loaded. See *Finding and running tests* for more.

Setting up your test environment is easier

nose supports fixtures at the package, module, class, and test case level, so expensive initialization can be done as infrequently as possible. See *Fixtures* for more.

Doing what you want to do is easier

nose comes with a number of *builtin plugins* to help you with output capture, error introspection, code coverage, doctests, and more. It also comes with plugin hooks for loading, running, watching and reporting on tests and test runs. If you don't like the default collection scheme, or it doesn't suit the layout of your project, or you need reports in a format different from the unittest standard, or you need to collect some additional information about tests (like code coverage or profiling data), you can write a plugin to make nose do what you want. See the section on *Writing Plugins* for more. There are also many third-party nose plugins available.

Details

Basic usage

Use the nosetests script (after installation by setuptools):

```
nosetests [options] [(optional) test files or directories]
```

In addition to passing command-line options, you may also put configuration options in a .noserc or nose.cfg file in your home directory. These are standard .ini-style config files. Put your nosetests configuration in a [nosetests] section, with the – prefix removed:

```
[nosetests]
verbosity=3
with-doctest=1
```

There is also possiblity to disable configuration files loading (might be useful when runnig i.e. tox and you don't want your global nose config file to be used by tox). In order to ignore those configuration files simply set an environment variable `NOSE_IGNORE_CONFIG_FILES`.

There are several other ways to use the nose test runner besides the *nosetests* script. You may use nose in a test script:

```
import nose
nose.main()
```

If you don't want the test script to exit with 0 on success and 1 on failure (like unittest.main), use nose.run() instead:

```
import nose
result = nose.run()
```

result will be true if the test run succeeded, or false if any test failed or raised an uncaught exception. Lastly, you can run nose.core directly, which will run nose.main():

```
python /path/to/nose/core.py
```

Please see the usage message for the nosetests script for information about how to control which tests nose runs, which plugins are loaded, and the test output.

Extended usage

nose collects tests automatically from python source files, directories and packages found in its working directory (which defaults to the current working directory). Any python source file, directory or package that matches the testMatch regular expression (by default: *(?:\b|_)[Tt]est* will be collected as a test (or source for collection of tests). In addition, all other packages found in the working directory will be examined for python source files or directories

that match testMatch. Package discovery descends all the way down the tree, so package.tests and package.sub.tests and package.sub.sub2.tests will all be collected.

Within a test directory or package, any python source file matching testMatch will be examined for test cases. Within a test module, functions and classes whose names match testMatch and TestCase subclasses with any name will be loaded and executed as tests. Tests may use the assert keyword or raise AssertionErrors to indicate test failure. TestCase subclasses may do the same or use the various TestCase methods available.

It is important to note that the default behavior of nose is to not include tests from files which are executable. To include tests from such files, remove their executable bit or use the –exe flag (see 'Options' section below).

Selecting Tests

To specify which tests to run, pass test names on the command line:

```
nosetests only_test_this.py
```

Test names specified may be file or module names, and may optionally indicate the test case to run by separating the module or file name from the test case name with a colon. Filenames may be relative or absolute. Examples:

```
nosetests test.module
nosetests another.test:TestCase.test_method
nosetests a.test:TestCase
nosetests /path/to/test/file.py:test_function
```

You may also change the working directory where nose looks for tests by using the -w switch:

```
nosetests -w /path/to/tests
```

Note, however, that support for multiple -w arguments is now deprecated and will be removed in a future release. As of nose 0.10, you can get the same behavior by specifying the target directories *without* the -w switch:

```
nosetests /path/to/tests /another/path/to/tests
```

Further customization of test selection and loading is possible through the use of plugins.

Test result output is identical to that of unittest, except for the additional features (error classes, and plugin-supplied features such as output capture and assert introspection) detailed in the options below.

Configuration

In addition to passing command-line options, you may also put configuration options in your project's *setup.cfg* file, or a .noserc or nose.cfg file in your home directory. In any of these standard ini-style config files, you put your nosetests configuration in a `[nosetests]` section. Options are the same as on the command line, with the – prefix removed. For options that are simple switches, you must supply a value:

```
[nosetests]
verbosity=3
with-doctest=1
```

All configuration files that are found will be loaded and their options combined. You can override the standard config file loading with the `-c` option.

Using Plugins

There are numerous nose plugins available via easy_install and elsewhere. To use a plugin, just install it. The plugin will add command line options to nosetests. To verify that the plugin is installed, run:

```
nosetests --plugins
```

You can add -v or -vv to that command to show more information about each plugin.

If you are running nose.main() or nose.run() from a script, you can specify a list of plugins to use by passing a list of plugins with the plugins keyword argument.

0.9 plugins

nose 1.0 can use SOME plugins that were written for nose 0.9. The default plugin manager inserts a compatibility wrapper around 0.9 plugins that adapts the changed plugin api calls. However, plugins that access nose internals are likely to fail, especially if they attempt to access test case or test suite classes. For example, plugins that try to determine if a test passed to startTest is an individual test or a suite will fail, partly because suites are no longer passed to startTest and partly because it's likely that the plugin is trying to find out if the test is an instance of a class that no longer exists.

0.10 and 0.11 plugins

All plugins written for nose 0.10 and 0.11 should work with nose 1.0.

Options

-V, --version
Output nose version and exit

-p, --plugins
Output list of available plugins and exit. Combine with higher verbosity for greater detail

-v=DEFAULT, **--verbose**=DEFAULT
Be more verbose. [NOSE_VERBOSE]

--verbosity=VERBOSITY
Set verbosity; –verbosity=2 is the same as -v

-q=DEFAULT, **--quiet**=DEFAULT
Be less verbose

-c=FILES, **--config**=FILES
Load configuration from config file(s). May be specified multiple times; in that case, all config files will be loaded and combined

-w=WHERE, **--where**=WHERE
Look for tests in this directory. May be specified multiple times. The first directory passed will be used as the working directory, in place of the current working directory, which is the default. Others will be added to the list of tests to execute. [NOSE_WHERE]

--py3where=PY3WHERE
Look for tests in this directory under Python 3.x. Functions the same as 'where', but only applies if running under Python 3.x or above. Note that, if present under 3.x, this option completely replaces any directories specified with 'where', so the 'where' option becomes ineffective. [NOSE_PY3WHERE]

-m=REGEX, **--match**=REGEX, **--testmatch**=REGEX
Files, directories, function names, and class names that match this regular expression are considered tests. Default: (?:\b|_)[Tt]est [NOSE_TESTMATCH]

--tests=NAMES
Run these tests (comma-separated list). This argument is useful mainly from configuration files; on the command line, just pass the tests to run as additional arguments with no switch.

-l=DEFAULT, **--debug**=DEFAULT
Activate debug logging for one or more systems. Available debug loggers: nose, nose.importer, nose.inspector, nose.plugins, nose.result and nose.selector. Separate multiple names with a comma.

--debug-log=FILE
Log debug messages to this file (default: sys.stderr)

--logging-config=FILE, **--log-config**=FILE
Load logging config from this file – bypasses all other logging config settings.

-I=REGEX, **--ignore-files**=REGEX
Completely ignore any file that matches this regular expression. Takes precedence over any other settings or plugins. Specifying this option will replace the default setting. Specify this option multiple times to add more regular expressions [NOSE_IGNORE_FILES]

-e=REGEX, **--exclude**=REGEX
Don't run tests that match regular expression [NOSE_EXCLUDE]

-i=REGEX, **--include**=REGEX
This regular expression will be applied to files, directories, function names, and class names for a chance to include additional tests that do not match TESTMATCH. Specify this option multiple times to add more regular expressions [NOSE_INCLUDE]

-x, --stop
Stop running tests after the first error or failure

-P, --no-path-adjustment
Don't make any changes to sys.path when loading tests [NOSE_NOPATH]

--exe
Look for tests in python modules that are executable. Normal behavior is to exclude executable modules, since they may not be import-safe [NOSE_INCLUDE_EXE]

--noexe
DO NOT look for tests in python modules that are executable. (The default on the windows platform is to do so.)

--traverse-namespace
Traverse through all path entries of a namespace package

--first-package-wins, --first-pkg-wins, --1st-pkg-wins
nose's importer will normally evict a package from sys.modules if it sees a package with the same name in a different location. Set this option to disable that behavior.

--no-byte-compile
Prevent nose from byte-compiling the source into .pyc files while nose is scanning for and running tests.

-a=ATTR, **--attr**=ATTR
Run only tests that have attributes specified by ATTR [NOSE_ATTR]

-A=EXPR, **--eval-attr**=EXPR
Run only tests for whose attributes the Python expression EXPR evaluates to True [NOSE_EVAL_ATTR]

-s, --nocapture
Don't capture stdout (any stdout output will be printed immediately) [NOSE_NOCAPTURE]

--nologcapture
Disable logging capture plugin. Logging configuration will be left intact. [NOSE_NOLOGCAPTURE]

--logging-format=FORMAT
Specify custom format to print statements. Uses the same format as used by standard logging handlers. [NOSE_LOGFORMAT]

--logging-datefmt=FORMAT
Specify custom date/time format to print statements. Uses the same format as used by standard logging handlers. [NOSE_LOGDATEFMT]

--logging-filter=FILTER
Specify which statements to filter in/out. By default, everything is captured. If the output is too verbose, use this option to filter out needless output. Example: filter=foo will capture statements issued ONLY to foo or foo.what.ever.sub but not foobar or other logger. Specify multiple loggers with comma: filter=foo,bar,baz. If any logger name is prefixed with a minus, eg filter=-foo, it will be excluded rather than included. Default: exclude logging messages from nose itself (-nose). [NOSE_LOGFILTER]

--logging-clear-handlers
Clear all other logging handlers

--logging-level=DEFAULT
Set the log level to capture

--with-coverage
Enable plugin Coverage: Activate a coverage report using Ned Batchelder's coverage module. [NOSE_WITH_COVERAGE]

--cover-package=PACKAGE
Restrict coverage output to selected packages [NOSE_COVER_PACKAGE]

--cover-erase
Erase previously collected coverage statistics before run

--cover-tests
Include test modules in coverage report [NOSE_COVER_TESTS]

--cover-min-percentage=DEFAULT
Minimum percentage of coverage for tests to pass [NOSE_COVER_MIN_PERCENTAGE]

--cover-inclusive
Include all python files under working directory in coverage report. Useful for discovering holes in test coverage if not all files are imported by the test suite. [NOSE_COVER_INCLUSIVE]

--cover-html
Produce HTML coverage information

--cover-html-dir=DIR
Produce HTML coverage information in dir

--cover-branches
Include branch coverage in coverage report [NOSE_COVER_BRANCHES]

--cover-xml
Produce XML coverage information

--cover-xml-file=FILE
Produce XML coverage information in file

--cover-config-file=DEFAULT
Location of coverage config file [NOSE_COVER_CONFIG_FILE]

--cover-no-print
Suppress printing of coverage information

--pdb
Drop into debugger on failures or errors

--pdb-failures
Drop into debugger on failures

--pdb-errors
Drop into debugger on errors

--no-deprecated
Disable special handling of DeprecatedTest exceptions.

--with-doctest
Enable plugin Doctest: Activate doctest plugin to find and run doctests in non-test modules. [NOSE_WITH_DOCTEST]

--doctest-tests
Also look for doctests in test modules. Note that classes, methods and functions should have either doctests or non-doctest tests, not both. [NOSE_DOCTEST_TESTS]

--doctest-extension=EXT
Also look for doctests in files with this extension [NOSE_DOCTEST_EXTENSION]

--doctest-result-variable=VAR
Change the variable name set to the result of the last interpreter command from the default '_'. Can be used to avoid conflicts with the _() function used for text translation. [NOSE_DOCTEST_RESULT_VAR]

--doctest-fixtures=SUFFIX
Find fixtures for a doctest file in module with this name appended to the base name of the doctest file

--doctest-options=OPTIONS
Specify options to pass to doctest. Eg. '+ELLIPSIS,+NORMALIZE_WHITESPACE'

--with-isolation
Enable plugin IsolationPlugin: Activate the isolation plugin to isolate changes to external modules to a single test module or package. The isolation plugin resets the contents of sys.modules after each test module or package runs to its state before the test. PLEASE NOTE that this plugin should not be used with the coverage plugin, or in any other case where module reloading may produce undesirable side-effects. [NOSE_WITH_ISOLATION]

-d, --detailed-errors, --failure-detail
Add detail to error output by attempting to evaluate failed asserts [NOSE_DETAILED_ERRORS]

--with-profile
Enable plugin Profile: Use this plugin to run tests using the hotshot profiler. [NOSE_WITH_PROFILE]

--profile-sort=SORT
Set sort order for profiler output

--profile-stats-file=FILE
Profiler stats file; default is a new temp file on each run

--profile-restrict=RESTRICT
Restrict profiler output. See help for pstats.Stats for details

--no-skip
Disable special handling of SkipTest exceptions.

--with-id
Enable plugin TestId: Activate to add a test id (like #1) to each test name output. Activate with –failed to rerun failing tests only. [NOSE_WITH_ID]

--id-file=FILE
Store test ids found in test runs in this file. Default is the file .noseids in the working directory.

--failed
Run the tests that failed in the last test run.

--processes=NUM
Spread test run among this many processes. Set a number equal to the number of processors or cores in your machine for best results. Pass a negative number to have the number of processes automatically set to the number of cores. Passing 0 means to disable parallel testing. Default is 0 unless NOSE_PROCESSES is set. [NOSE_PROCESSES]

--process-timeout=SECONDS
Set timeout for return of results from each test runner process. Default is 10. [NOSE_PROCESS_TIMEOUT]

--process-restartworker
If set, will restart each worker process once their tests are done, this helps control memory leaks from killing the system. [NOSE_PROCESS_RESTARTWORKER]

--with-xunit
Enable plugin Xunit: This plugin provides test results in the standard XUnit XML format. [NOSE_WITH_XUNIT]

--xunit-file=FILE
Path to xml file to store the xunit report in. Default is nosetests.xml in the working directory [NOSE_XUNIT_FILE]

--xunit-testsuite-name=PACKAGE
Name of the testsuite in the xunit xml, generated by plugin. Default test suite name is nosetests.

--xunit-prefix-with-testsuite-name
Whether to prefix the class name under test with testsuite name. Defaults to false.

--all-modules
Enable plugin AllModules: Collect tests from all python modules. [NOSE_ALL_MODULES]

--collect-only
Enable collect-only: Collect and output test names only, don't run any tests. [COLLECT_ONLY]

Writing tests

As with py.test, nose tests need not be subclasses of `unittest.TestCase`. Any function or class that matches the configured testMatch regular expression (`(?:^|[\\b_\\.-])[Tt]est` by default – that is, has test or Test at a word boundary or following a - or _) and lives in a module that also matches that expression will be run as a test. For the sake of compatibility with legacy unittest test cases, nose will also load tests from `unittest.TestCase` subclasses just like unittest does. Like py.test, nose runs functional tests in the order in which they appear in the module file. TestCase-derived tests and other test classes are run in alphabetical order.

Fixtures

nose supports fixtures (setup and teardown methods) at the package, module, class, and test level. As with py.test or unittest fixtures, setup always runs before any test (or collection of tests for test packages and modules); teardown runs if setup has completed successfully, regardless of the status of the test run. For more detail on fixtures at each level, see below.

Test packages

nose allows tests to be grouped into test packages. This allows package-level setup; for instance, if you need to create a test database or other data fixture for your tests, you may create it in package setup and remove it in package teardown once per test run, rather than having to create and tear it down once per test module or test case.

To create package-level setup and teardown methods, define setup and/or teardown functions in the `__init__.py` of a test package. Setup methods may be named *setup*, *setup_package*, *setUp*, or *setUpPackage*; teardown may be named *teardown*, *teardown_package*, *tearDown* or *tearDownPackage*. Execution of tests in a test package begins as soon as the first test module is loaded from the test package.

Test modules

A test module is a python module that matches the testMatch regular expression. Test modules offer module-level setup and teardown; define the method *setup*, *setup_module*, *setUp* or *setUpModule* for setup, *teardown*, *teardown_module*, or *tearDownModule* for teardown. Execution of tests in a test module begins after all tests are collected.

Test classes

A test class is a class defined in a test module that matches testMatch or is a subclass of `unittest.TestCase`. All test classes are run the same way: Methods in the class that match testMatch are discovered, and a test case is constructed to run each method with a fresh instance of the test class. Like `unittest.TestCase` subclasses, other test classes can define setUp and tearDown methods that will be run before and after each test method. Test classes that do not descend from *unittest.TestCase* may also include generator methods and class-level fixtures. Class-level setup fixtures may be named *setup_class*, *setupClass*, *setUpClass*, *setupAll* or *setUpAll*; teardown fixtures may be named *teardown_class*, *teardownClass*, *tearDownClass*, *teardownAll* or *tearDownAll*. Class-level setup and teardown fixtures must be class methods.

Test functions

Any function in a test module that matches testMatch will be wrapped in a *FunctionTestCase* and run as a test. The simplest possible failing test is therefore:

```
def test():
    assert False
```

And the simplest passing test:

```
def test():
    pass
```

Test functions may define setup and/or teardown attributes, which will be run before and after the test function, respectively. A convenient way to do this, especially when several test functions in the same module need the same setup, is to use the provided *with_setup* decorator:

```
def setup_func():
    "set up test fixtures"

def teardown_func():
    "tear down test fixtures"

@with_setup(setup_func, teardown_func)
```

```
def test():
    "test ..."
```

For python 2.3 or earlier, add the attributes by calling the decorator function like so:

```
def test():
    "test ... "
test = with_setup(setup_func, teardown_func)(test)
```

or by direct assignment:

```
test.setup = setup_func
test.teardown = teardown_func
```

Please note that *with_setup* is useful *only* for test functions, not for test methods in *unittest.TestCase* subclasses or other test classes. For those cases, define *setUp* and *tearDown* methods in the class.

Test generators

nose supports test functions and methods that are generators. A simple example from nose's selftest suite is probably the best explanation:

```
def test_evens():
    for i in range(0, 5):
        yield check_even, i, i*3

def check_even(n, nn):
    assert n % 2 == 0 or nn % 2 == 0
```

This will result in five tests. nose will iterate the generator, creating a function test case wrapper for each tuple it yields. As in the example, test generators must yield tuples, the first element of which must be a callable and the remaining elements the arguments to be passed to the callable.

By default, the test name output for a generated test in verbose mode will be the name of the generator function or method, followed by the args passed to the yielded callable. If you want to show a different test name, set the `description` attribute of the yielded callable.

Setup and teardown functions may be used with test generators. However, please note that setup and teardown attributes attached to the *generator function* will execute only once. To *execute fixtures for each yielded test*, attach the setup and teardown attributes to the function that is yielded, or yield a callable object instance with setup and teardown attributes.

For example:

```
@with_setup(setup_func, teardown_func)
def test_generator():
    # ...
    yield func, arg, arg # ...
```

Here, the setup and teardown functions will be executed *once*. Compare to:

```
def test_generator():
    # ...
    yield func, arg, arg # ...

@with_setup(setup_func, teardown_func)
```

```
def func(arg):
    assert something_about(arg)
```

In the latter case the setup and teardown functions will execute once for each yielded test.

For generator methods, the setUp and tearDown methods of the class (if any) will be run before and after each generated test case. The setUp and tearDown methods *do not* run before the generator method itself, as this would cause setUp to run twice before the first test without an intervening tearDown.

Please note that method generators *are not* supported in *unittest.TestCase* subclasses.

Finding and running tests

nose, by default, follows a few simple rules for test discovery.

- If it looks like a test, it's a test. Names of directories, modules, classes and functions are compared against the testMatch regular expression, and those that match are considered tests. Any class that is a *unittest.TestCase* subclass is also collected, so long as it is inside of a module that looks like a test.
- Files with the executable bit set are ignored by default under Unix-style operating systems–use `--exe` to allow collection from them, but be careful that is safe to do so. Under Windows, executable files will be picked up by default since there is no executable bit to test.
- Directories that don't look like tests and aren't packages are not inspected.
- Packages are always inspected, but they are only collected if they look like tests. This means that you can include your tests inside of your packages (somepackage/tests) and nose will collect the tests without running package code inappropriately.
- When a project appears to have library and test code organized into separate directories, library directories are examined first.
- When nose imports a module, it adds that module's directory to sys.path; when the module is inside of a package, like package.module, it will be loaded as package.module and the directory of *package* will be added to sys.path.
- If an object defines a __test__ attribute that does not evaluate to True, that object will not be collected, nor will any objects it contains.

Be aware that plugins and command line options can change any of those rules.

Testing tools

The nose.tools module provides a number of testing aids that you may find useful, including decorators for restricting test execution time and testing for exceptions, and all of the same assertX methods found in *unittest.TestCase* (only spelled in **PEP 8#function-names** fashion, so *assert_equal* rather than *assertEqual*).

Tools for testing

nose.tools provides a few convenience functions to make writing tests easier. You don't have to use them; nothing in the rest of nose depends on any of these methods.

`nose.tools.ok_`(*expr*, *msg=None*)
Shorthand for assert. Saves 3 whole characters!

`nose.tools.eq_`(*a*, *b*, *msg=None*)
Shorthand for 'assert a == b, "%r != %r" % (a, b)

nose.tools.**make_decorator**(*func*)
Wraps a test decorator so as to properly replicate metadata of the decorated function, including nose's additional stuff (namely, setup and teardown).

nose.tools.**raises**(**exceptions*)
Test must raise one of expected exceptions to pass.

Example use:

```
@raises(TypeError, ValueError)
def test_raises_type_error():
    raise TypeError("This test passes")

@raises(Exception)
def test_that_fails_by_passing():
    pass
```

If you want to test many assertions about exceptions in a single test, you may want to use *assert_raises* instead.

nose.tools.**set_trace**()
Call pdb.set_trace in the calling frame, first restoring sys.stdout to the real output stream. Note that sys.stdout is NOT reset to whatever it was before the call once pdb is done!

nose.tools.**timed**(*limit*)
Test must finish within specified time limit to pass.

Example use:

```
@timed(.1)
def test_that_fails():
    time.sleep(.2)
```

nose.tools.**with_setup**(*setup=None*, *teardown=None*)
Decorator to add setup and/or teardown methods to a test function:

```
@with_setup(setup, teardown)
def test_something():
    " ... "
```

Note that *with_setup* is useful *only* for test functions, not for test methods or inside of TestCase subclasses.

nose.tools.**istest**(*func*)
Decorator to mark a function or method as a test

nose.tools.**nottest**(*func*)
Decorator to mark a function or method as *not* a test

Batteries included: builtin nose plugins

nose includes a number of builtin plugins that can make testing faster and easier.

Note: nose 0.11.2 includes a change to default plugin loading. Now, a 3rd party plugin with *the same name* as a builtin *will be loaded instead* of the builtin plugin.

AllModules: collect tests in all modules

Use the AllModules plugin by passing `--all-modules` or setting the NOSE_ALL_MODULES environment variable to enable collection and execution of tests in all python modules. Normal nose behavior is to look for tests only in modules that match testMatch.

More information: *Finding tests in all modules*

> **Warning:** This plugin can have surprising interactions with plugins that load tests from what nose normally considers non-test modules, such as the *doctest plugin*. This is because any given object in a module can't be loaded both by a plugin and the normal nose *test loader*. Also, if you have functions or classes in non-test modules that look like tests but aren't, you will likely see errors as nose attempts to run them as tests.

Options

--all-modules
Enable plugin AllModules: Collect tests from all python modules. [NOSE_ALL_MODULES]

Plugin

class nose.plugins.allmodules.**AllModules**
Bases: *nose.plugins.base.Plugin*

Collect tests from all python modules.

options (*parser*, *env*)
Register commandline options.

wantFile (*file*)
Override to return True for all files ending with .py

wantModule (*module*)
Override return True for all modules

Source

```
"""Use the AllModules plugin by passing ``--all-modules`` or setting the
NOSE_ALL_MODULES environment variable to enable collection and execution of
tests in all python modules. Normal nose behavior is to look for tests only in
modules that match testMatch.

More information: :doc:`../doc_tests/test_allmodules/test_allmodules`

.. warning ::

   This plugin can have surprising interactions with plugins that load tests
   from what nose normally considers non-test modules, such as
   the :doc:`doctest plugin <doctests>`. This is because any given
   object in a module can't be loaded both by a plugin and the normal nose
   :class:`test loader <nose.loader.TestLoader>`. Also, if you have functions
   or classes in non-test modules that look like tests but aren't, you will
   likely see errors as nose attempts to run them as tests.
```

```
"""

import os
from nose.plugins.base import Plugin

class AllModules(Plugin):
    """Collect tests from all python modules.
    """
    def options(self, parser, env):
        """Register commandline options.
        """
        env_opt = 'NOSE_ALL_MODULES'
        parser.add_option('--all-modules',
                          action="store_true",
                          dest=self.enableOpt,
                          default=env.get(env_opt),
                          help="Enable plugin %s: %s [%s]" %
                          (self.__class__.__name__, self.help(), env_opt))

    def wantFile(self, file):
        """Override to return True for all files ending with .py"""
        # always want .py files
        if file.endswith('.py'):
            return True

    def wantModule(self, module):
        """Override return True for all modules"""
        return True
```

Attrib: tag and select tests with attributes

Attribute selector plugin.

Oftentimes when testing you will want to select tests based on criteria rather then simply by filename. For example, you might want to run all tests except for the slow ones. You can do this with the Attribute selector plugin by setting attributes on your test methods. Here is an example:

```
def test_big_download():
    import urllib
    # commence slowness...

test_big_download.slow = 1
```

Once you've assigned an attribute `slow = 1` you can exclude that test and all other tests having the slow attribute by running

```
$ nosetests -a '!slow'
```

There is also a decorator available for you that will set attributes. Here's how to set `slow=1` like above with the decorator:

```
from nose.plugins.attrib import attr
@attr('slow')
def test_big_download():
    import urllib
    # commence slowness...
```

And here's how to set an attribute with a specific value:

```
from nose.plugins.attrib import attr
@attr(speed='slow')
def test_big_download():
    import urllib
    # commence slowness...
```

This test could be run with

```
$ nosetests -a speed=slow
```

In Python 2.6 and higher, `@attr` can be used on a class to set attributes on all its test methods at once. For example:

```
from nose.plugins.attrib import attr
@attr(speed='slow')
class MyTestCase:
    def test_long_integration(self):
        pass
    def test_end_to_end_something(self):
        pass
```

Below is a reference to the different syntaxes available.

Simple syntax

Examples of using the `-a` and `--attr` options:

- **`nosetests -a status=stable`** Only runs tests with attribute "status" having value "stable"
- **`nosetests -a priority=2,status=stable`** Runs tests having both attributes and values
- **`nosetests -a priority=2 -a slow`** Runs tests that match either attribute
- **`nosetests -a tags=http`** If a test's `tags` attribute was a list and it contained the value `http` then it would be run
- **`nosetests -a slow`** Runs tests with the attribute `slow` if its value does not equal False (False, [], "", etc...)
- **`nosetests -a '!slow'`** Runs tests that do NOT have the attribute `slow` or have a `slow` attribute that is equal to False **NOTE**: if your shell (like bash) interprets '!' as a special character make sure to put single quotes around it.

Expression Evaluation

Examples using the `-A` and `--eval-attr` options:

- `nosetests -A "not slow"` Evaluates the Python expression "not slow" and runs the test if True
- `nosetests -A "(priority > 5) and not slow"` Evaluates a complex Python expression and runs the test if True

Options

`-a``=ATTR`, **`--attr`**`=ATTR`
Run only tests that have attributes specified by ATTR [NOSE_ATTR]

-A=EXPR, **--eval-attr**=EXPR
Run only tests for whose attributes the Python expression EXPR evaluates to True [NOSE_EVAL_ATTR]

Plugin

class nose.plugins.attrib.**AttributeSelector**
Bases: *nose.plugins.base.Plugin*

Selects test cases to be run based on their attributes.

configure(*options*, *config*)
Configure the plugin and system, based on selected options.

attr and eval_attr may each be lists.

self.attribs will be a list of lists of tuples. In that list, each list is a group of attributes, all of which must match for the rule to match.

options(*parser*, *env*)
Register command line options

validateAttrib(*method*, *cls=None*)
Verify whether a method has the required attributes The method is considered a match if it matches all attributes for any attribute group. .

wantFunction(*function*)
Accept the function if its attributes match.

wantMethod(*method*)
Accept the method if its attributes match.

Source

```
"""Attribute selector plugin.

Oftentimes when testing you will want to select tests based on
criteria rather then simply by filename. For example, you might want
to run all tests except for the slow ones. You can do this with the
Attribute selector plugin by setting attributes on your test methods.
Here is an example:

.. code-block:: python

    def test_big_download():
        import urllib
        # commence slowness...

    test_big_download.slow = 1

Once you've assigned an attribute ``slow = 1`` you can exclude that
test and all other tests having the slow attribute by running ::

    $ nosetests -a '!slow'

There is also a decorator available for you that will set attributes.
Here's how to set ``slow=1`` like above with the decorator:
```

```
.. code-block:: python

    from nose.plugins.attrib import attr
    @attr('slow')
    def test_big_download():
        import urllib
        # commence slowness...

And here's how to set an attribute with a specific value:

.. code-block:: python

    from nose.plugins.attrib import attr
    @attr(speed='slow')
    def test_big_download():
        import urllib
        # commence slowness...

This test could be run with ::

    $ nosetests -a speed=slow

In Python 2.6 and higher, ``@attr`` can be used on a class to set attributes
on all its test methods at once.  For example:

.. code-block:: python

    from nose.plugins.attrib import attr
    @attr(speed='slow')
    class MyTestCase:
        def test_long_integration(self):
            pass
        def test_end_to_end_something(self):
            pass

Below is a reference to the different syntaxes available.

Simple syntax
-------------

Examples of using the ``-a`` and ``--attr`` options:

* ``nosetests -a status=stable``
   Only runs tests with attribute "status" having value "stable"

* ``nosetests -a priority=2,status=stable``
   Runs tests having both attributes and values

* ``nosetests -a priority=2 -a slow``
   Runs tests that match either attribute

* ``nosetests -a tags=http``
   If a test's ``tags`` attribute was a list and it contained the value
   ``http`` then it would be run

* ``nosetests -a slow``
   Runs tests with the attribute ``slow`` if its value does not equal False
   (False, [], "", etc...)
```

```
* ``nosetests -a '!slow'``
   Runs tests that do NOT have the attribute ``slow`` or have a ``slow``
   attribute that is equal to False
   **NOTE**:
   if your shell (like bash) interprets '!' as a special character make sure to
   put single quotes around it.

Expression Evaluation
---------------------

Examples using the ``-A`` and ``--eval-attr`` options:

* ``nosetests -A "not slow"``
  Evaluates the Python expression "not slow" and runs the test if True

* ``nosetests -A "(priority > 5) and not slow"``
  Evaluates a complex Python expression and runs the test if True

"""
import inspect
import logging
import os
import sys
from inspect import isfunction
from nose.plugins.base import Plugin
from nose.util import tolist

log = logging.getLogger('nose.plugins.attrib')
compat_24 = sys.version_info >= (2, 4)

def attr(*args, **kwargs):
    """Decorator that adds attributes to classes or functions
    for use with the Attribute (-a) plugin.
    """
    def wrap_ob(ob):
        for name in args:
            setattr(ob, name, True)
        for name, value in kwargs.iteritems():
            setattr(ob, name, value)
        return ob
    return wrap_ob

def get_method_attr(method, cls, attr_name, default = False):
    """Look up an attribute on a method/ function.
    If the attribute isn't found there, looking it up in the
    method's class, if any.
    """
    Missing = object()
    value = getattr(method, attr_name, Missing)
    if value is Missing and cls is not None:
        value = getattr(cls, attr_name, Missing)
    if value is Missing:
        return default
    return value


class ContextHelper:
```

```
    """Object that can act as context dictionary for eval and looks up
    names as attributes on a method/ function and its class.
    """
    def __init__(self, method, cls):
        self.method = method
        self.cls = cls

    def __getitem__(self, name):
        return get_method_attr(self.method, self.cls, name)


class AttributeSelector(Plugin):
    """Selects test cases to be run based on their attributes.
    """

    def __init__(self):
        Plugin.__init__(self)
        self.attribs = []

    def options(self, parser, env):
        """Register command line options"""
        parser.add_option("-a", "--attr",
                          dest="attr", action="append",
                          default=env.get('NOSE_ATTR'),
                          metavar="ATTR",
                          help="Run only tests that have attributes "
                          "specified by ATTR [NOSE_ATTR]")
        # disable in < 2.4: eval can't take needed args
        if compat_24:
            parser.add_option("-A", "--eval-attr",
                              dest="eval_attr", metavar="EXPR", action="append",
                              default=env.get('NOSE_EVAL_ATTR'),
                              help="Run only tests for whose attributes "
                              "the Python expression EXPR evaluates "
                              "to True [NOSE_EVAL_ATTR]")

    def configure(self, options, config):
        """Configure the plugin and system, based on selected options.

        attr and eval_attr may each be lists.

        self.attribs will be a list of lists of tuples. In that list, each
        list is a group of attributes, all of which must match for the rule to
        match.
        """
        self.attribs = []

        # handle python eval-expression parameter
        if compat_24 and options.eval_attr:
            eval_attr = tolist(options.eval_attr)
            for attr in eval_attr:
                # "<python expression>"
                # -> eval(expr) in attribute context must be True
                def eval_in_context(expr, obj, cls):
                    return eval(expr, None, ContextHelper(obj, cls))
                self.attribs.append([(attr, eval_in_context)])

        # attribute requirements are a comma separated list of
```

```
        # 'key=value' pairs
        if options.attr:
            std_attr = tolist(options.attr)
            for attr in std_attr:
                # all attributes within an attribute group must match
                attr_group = []
                for attrib in attr.strip().split(","):
                    # don't die on trailing comma
                    if not attrib:
                        continue
                    items = attrib.split("=", 1)
                    if len(items) > 1:
                        # "name=value"
                        # -> 'str(obj.name) == value' must be True
                        key, value = items
                    else:
                        key = items[0]
                        if key[0] == "!":
                            # "!name"
                            # 'bool(obj.name)' must be False
                            key = key[1:]
                            value = False
                        else:
                            # "name"
                            # -> 'bool(obj.name)' must be True
                            value = True
                    attr_group.append((key, value))
                self.attribs.append(attr_group)
        if self.attribs:
            self.enabled = True

    def validateAttrib(self, method, cls = None):
        """Verify whether a method has the required attributes
        The method is considered a match if it matches all attributes
        for any attribute group.
        ."""
        # TODO: is there a need for case-sensitive value comparison?
        any = False
        for group in self.attribs:
            match = True
            for key, value in group:
                attr = get_method_attr(method, cls, key)
                if callable(value):
                    if not value(key, method, cls):
                        match = False
                        break
                elif value is True:
                    # value must exist and be True
                    if not bool(attr):
                        match = False
                        break
                elif value is False:
                    # value must not exist or be False
                    if bool(attr):
                        match = False
                        break
                elif type(attr) in (list, tuple):
                    # value must be found in the list attribute
```

```
                    if not str(value).lower() in [str(x).lower()
                                                  for x in attr]:
                        match = False
                        break
                else:
                    # value must match, convert to string and compare
                    if (value != attr
                        and str(value).lower() != str(attr).lower()):
                        match = False
                        break
            any = any or match
        if any:
            # not True because we don't want to FORCE the selection of the
            # item, only say that it is acceptable
            return None
        return False

    def wantFunction(self, function):
        """Accept the function if its attributes match.
        """
        return self.validateAttrib(function)

    def wantMethod(self, method):
        """Accept the method if its attributes match.
        """
        try:
            cls = method.im_class
        except AttributeError:
            return False
        return self.validateAttrib(method, cls)
```

Capture: capture stdout during tests

This plugin captures stdout during test execution. If the test fails or raises an error, the captured output will be appended to the error or failure output. It is enabled by default but can be disabled with the options `-s` or `--nocapture`.

Options

`--nocapture` Don't capture stdout (any stdout output will be printed immediately)

Options

`-s, --nocapture`
Don't capture stdout (any stdout output will be printed immediately) [NOSE_NOCAPTURE]

Plugin

class `nose.plugins.capture.`**`Capture`**
Bases: *nose.plugins.base.Plugin*

Output capture plugin. Enabled by default. Disable with `-s` or `--nocapture`. This plugin captures stdout during test execution, appending any output captured to the error or failure output, should the test fail or raise an error.

afterTest (*test*)
Clear capture buffer.

beforeTest (*test*)
Flush capture buffer.

begin ()
Replace sys.stdout with capture buffer.

buffer
Captured stdout output.

configure (*options*, *conf*)
Configure plugin. Plugin is enabled by default.

finalize (*result*)
Restore stdout.

formatError (*test*, *err*)
Add captured output to error report.

formatFailure (*test*, *err*)
Add captured output to failure report.

options (*parser*, *env*)
Register commandline options

Source

```
"""
This plugin captures stdout during test execution. If the test fails
or raises an error, the captured output will be appended to the error
or failure output. It is enabled by default but can be disabled with
the options ``-s`` or ``--nocapture``.

:Options:
  ``--nocapture``
    Don't capture stdout (any stdout output will be printed immediately)

"""
import logging
import os
import sys
from nose.plugins.base import Plugin
from nose.pyversion import exc_to_unicode, force_unicode
from nose.util import ln
from StringIO import StringIO


log = logging.getLogger(__name__)

class Capture(Plugin):
    """
    Output capture plugin. Enabled by default. Disable with ``-s`` or
    ``--nocapture``. This plugin captures stdout during test execution,
    appending any output captured to the error or failure output,
    should the test fail or raise an error.
    """
    enabled = True
```

```
    env_opt = 'NOSE_NOCAPTURE'
    name = 'capture'
    score = 1600

    def __init__(self):
        self.stdout = []
        self._buf = None

    def options(self, parser, env):
        """Register commandline options
        """
        parser.add_option(
            "-s", "--nocapture", action="store_false",
            default=not env.get(self.env_opt), dest="capture",
            help="Don't capture stdout (any stdout output "
            "will be printed immediately) [NOSE_NOCAPTURE]")

    def configure(self, options, conf):
        """Configure plugin. Plugin is enabled by default.
        """
        self.conf = conf
        if not options.capture:
            self.enabled = False

    def afterTest(self, test):
        """Clear capture buffer.
        """
        self.end()
        self._buf = None

    def begin(self):
        """Replace sys.stdout with capture buffer.
        """
        self.start() # get an early handle on sys.stdout

    def beforeTest(self, test):
        """Flush capture buffer.
        """
        self.start()

    def formatError(self, test, err):
        """Add captured output to error report.
        """
        test.capturedOutput = output = self.buffer
        self._buf = None
        if not output:
            # Don't return None as that will prevent other
            # formatters from formatting and remove earlier formatters
            # formats, instead return the err we got
            return err
        ec, ev, tb = err
        return (ec, self.addCaptureToErr(ev, output), tb)

    def formatFailure(self, test, err):
        """Add captured output to failure report.
        """
        return self.formatError(test, err)
```

```
    def addCaptureToErr(self, ev, output):
        ev = exc_to_unicode(ev)
        output = force_unicode(output)
        return u'\n'.join([ev, ln(u'>> begin captured stdout <<'),
                           output, ln(u'>> end captured stdout <<')])

    def start(self):
        self.stdout.append(sys.stdout)
        self._buf = StringIO()
        # Python 3's StringIO objects don't support setting encoding or errors
        # directly and they're already set to None.  So if the attributes
        # already exist, skip adding them.
        if (not hasattr(self._buf, 'encoding') and
                hasattr(sys.stdout, 'encoding')):
            self._buf.encoding = sys.stdout.encoding
        if (not hasattr(self._buf, 'errors') and
                hasattr(sys.stdout, 'errors')):
            self._buf.errors = sys.stdout.errors
        sys.stdout = self._buf

    def end(self):
        if self.stdout:
            sys.stdout = self.stdout.pop()

    def finalize(self, result):
        """Restore stdout.
        """
        while self.stdout:
            self.end()

    def _get_buffer(self):
        if self._buf is not None:
            return self._buf.getvalue()

    buffer = property(_get_buffer, None, None,
                      """Captured stdout output.""")
```

Collect: Collect tests quickly

This plugin bypasses the actual execution of tests, and instead just collects test names. Fixtures are also bypassed, so running nosetests with the collection plugin enabled should be very quick.

This plugin is useful in combination with the testid plugin (`--with-id`). Run both together to get an indexed list of all tests, which will enable you to run individual tests by index number.

This plugin is also useful for counting tests in a test suite, and making people watching your demo think all of your tests pass.

Options

`--collect-only`
: Enable collect-only: Collect and output test names only, don't run any tests. [COLLECT_ONLY]

Plugin

class nose.plugins.collect.**CollectOnly**
Bases: *nose.plugins.base.Plugin*

Collect and output test names only, don't run any tests.

options(*parser*, *env*)
Register commandline options.

prepareTestCase(*test*)
Replace actual test with dummy that always passes.

prepareTestLoader(*loader*)
Install collect-only suite class in TestLoader.

Source

```
"""
This plugin bypasses the actual execution of tests, and instead just collects
test names. Fixtures are also bypassed, so running nosetests with the
collection plugin enabled should be very quick.

This plugin is useful in combination with the testid plugin (``--with-id``).
Run both together to get an indexed list of all tests, which will enable you to
run individual tests by index number.

This plugin is also useful for counting tests in a test suite, and making
people watching your demo think all of your tests pass.
"""
from nose.plugins.base import Plugin
from nose.case import Test
import logging
import unittest

log = logging.getLogger(__name__)


class CollectOnly(Plugin):
    """
    Collect and output test names only, don't run any tests.
    """
    name = "collect-only"
    enableOpt = 'collect_only'

    def options(self, parser, env):
        """Register commandline options.
        """
        parser.add_option('--collect-only',
                          action='store_true',
                          dest=self.enableOpt,
                          default=env.get('NOSE_COLLECT_ONLY'),
                          help="Enable collect-only: %s [COLLECT_ONLY]" %
                          (self.help()))

    def prepareTestLoader(self, loader):
        """Install collect-only suite class in TestLoader.
```

```
        """
        # Disable context awareness
        log.debug("Preparing test loader")
        loader.suiteClass = TestSuiteFactory(self.conf)

    def prepareTestCase(self, test):
        """Replace actual test with dummy that always passes.
        """
        # Return something that always passes
        log.debug("Preparing test case %s", test)
        if not isinstance(test, Test):
            return
        def run(result):
            # We need to make these plugin calls because there won't be
            # a result proxy, due to using a stripped-down test suite
            self.conf.plugins.startTest(test)
            result.startTest(test)
            self.conf.plugins.addSuccess(test)
            result.addSuccess(test)
            self.conf.plugins.stopTest(test)
            result.stopTest(test)
        return run


class TestSuiteFactory:
    """
    Factory for producing configured test suites.
    """
    def __init__(self, conf):
        self.conf = conf

    def __call__(self, tests=(), **kw):
        return TestSuite(tests, conf=self.conf)


class TestSuite(unittest.TestSuite):
    """
    Basic test suite that bypasses most proxy and plugin calls, but does
    wrap tests in a nose.case.Test so prepareTestCase will be called.
    """
    def __init__(self, tests=(), conf=None):
        self.conf = conf
        # Exec lazy suites: makes discovery depth-first
        if callable(tests):
            tests = tests()
        log.debug("TestSuite(%r)", tests)
        unittest.TestSuite.__init__(self, tests)

    def addTest(self, test):
        log.debug("Add test %s", test)
        if isinstance(test, unittest.TestSuite):
            self._tests.append(test)
        else:
            self._tests.append(Test(test, config=self.conf))
```

Cover: code coverage

Note: Newer versions of coverage contain their own nose plugin which is superior to the builtin plugin. It exposes more of coverage's options and uses coverage's native html output. Depending on the version of coverage installed, the included plugin may override the nose builtin plugin, or be available under a different name. Check `nosetests --help` or `nosetests --plugins` to find out which coverage plugin is available on your system.

If you have Ned Batchelder's coverage module installed, you may activate a coverage report with the `--with-coverage` switch or NOSE_WITH_COVERAGE environment variable. The coverage report will cover any python source module imported after the start of the test run, excluding modules that match testMatch. If you want to include those modules too, use the `--cover-tests` switch, or set the NOSE_COVER_TESTS environment variable to a true value. To restrict the coverage report to modules from a particular package or packages, use the `--cover-package` switch or the NOSE_COVER_PACKAGE environment variable.

Options

`--with-coverage`
Enable plugin Coverage: Activate a coverage report using Ned Batchelder's coverage module. [NOSE_WITH_COVERAGE]

`--cover-package``=PACKAGE`
Restrict coverage output to selected packages [NOSE_COVER_PACKAGE]

`--cover-erase`
Erase previously collected coverage statistics before run

`--cover-tests`
Include test modules in coverage report [NOSE_COVER_TESTS]

`--cover-min-percentage``=DEFAULT`
Minimum percentage of coverage for tests to pass [NOSE_COVER_MIN_PERCENTAGE]

`--cover-inclusive`
Include all python files under working directory in coverage report. Useful for discovering holes in test coverage if not all files are imported by the test suite. [NOSE_COVER_INCLUSIVE]

`--cover-html`
Produce HTML coverage information

`--cover-html-dir``=DIR`
Produce HTML coverage information in dir

`--cover-branches`
Include branch coverage in coverage report [NOSE_COVER_BRANCHES]

`--cover-xml`
Produce XML coverage information

`--cover-xml-file``=FILE`
Produce XML coverage information in file

`--cover-config-file``=DEFAULT`
Location of coverage config file [NOSE_COVER_CONFIG_FILE]

`--cover-no-print`
Suppress printing of coverage information

Plugin

class nose.plugins.cover.**Coverage**

Bases: *nose.plugins.base.Plugin*

Activate a coverage report using Ned Batchelder's coverage module.

afterTest(**args*, ***kwargs*)
Stop recording coverage information.

beforeTest(**args*, ***kwargs*)
Begin recording coverage information.

configure(*options*, *conf*)
Configure plugin.

options(*parser*, *env*)
Add options to command line.

report(*stream*)
Output code coverage report.

wantFile(*file*, *package=None*)
If inclusive coverage enabled, return true for all source files in wanted packages.

Source

```
"""If you have Ned Batchelder's coverage_ module installed, you may activate a
coverage report with the ``--with-coverage`` switch or NOSE_WITH_COVERAGE
environment variable. The coverage report will cover any python source module
imported after the start of the test run, excluding modules that match
testMatch. If you want to include those modules too, use the ``--cover-tests``
switch, or set the NOSE_COVER_TESTS environment variable to a true value. To
restrict the coverage report to modules from a particular package or packages,
use the ``--cover-package`` switch or the NOSE_COVER_PACKAGE environment
variable.

.. _coverage: http://www.nedbatchelder.com/code/modules/coverage.html
"""
import logging
import re
import sys
import StringIO
from nose.plugins.base import Plugin
from nose.util import src, tolist

log = logging.getLogger(__name__)


class Coverage(Plugin):
    """
    Activate a coverage report using Ned Batchelder's coverage module.
    """
    coverTests = False
    coverPackages = None
    coverInstance = None
    coverErase = False
    coverMinPercentage = None
```

```
    coverPrint = True
    score = 200
    status = {}

    def options(self, parser, env):
        """
        Add options to command line.
        """
        super(Coverage, self).options(parser, env)
        parser.add_option("--cover-package", action="append",
                          default=env.get('NOSE_COVER_PACKAGE'),
                          metavar="PACKAGE",
                          dest="cover_packages",
                          help="Restrict coverage output to selected packages "
                          "[NOSE_COVER_PACKAGE]")
        parser.add_option("--cover-erase", action="store_true",
                          default=env.get('NOSE_COVER_ERASE'),
                          dest="cover_erase",
                          help="Erase previously collected coverage "
                          "statistics before run")
        parser.add_option("--cover-tests", action="store_true",
                          dest="cover_tests",
                          default=env.get('NOSE_COVER_TESTS'),
                          help="Include test modules in coverage report "
                          "[NOSE_COVER_TESTS]")
        parser.add_option("--cover-min-percentage", action="store",
                          dest="cover_min_percentage",
                          default=env.get('NOSE_COVER_MIN_PERCENTAGE'),
                          help="Minimum percentage of coverage for tests "
                          "to pass [NOSE_COVER_MIN_PERCENTAGE]")
        parser.add_option("--cover-inclusive", action="store_true",
                          dest="cover_inclusive",
                          default=env.get('NOSE_COVER_INCLUSIVE'),
                          help="Include all python files under working "
                          "directory in coverage report.  Useful for "
                          "discovering holes in test coverage if not all "
                          "files are imported by the test suite. "
                          "[NOSE_COVER_INCLUSIVE]")
        parser.add_option("--cover-html", action="store_true",
                          default=env.get('NOSE_COVER_HTML'),
                          dest='cover_html',
                          help="Produce HTML coverage information")
        parser.add_option('--cover-html-dir', action='store',
                          default=env.get('NOSE_COVER_HTML_DIR', 'cover'),
                          dest='cover_html_dir',
                          metavar='DIR',
                          help='Produce HTML coverage information in dir')
        parser.add_option("--cover-branches", action="store_true",
                          default=env.get('NOSE_COVER_BRANCHES'),
                          dest="cover_branches",
                          help="Include branch coverage in coverage report "
                          "[NOSE_COVER_BRANCHES]")
        parser.add_option("--cover-xml", action="store_true",
                          default=env.get('NOSE_COVER_XML'),
                          dest="cover_xml",
                          help="Produce XML coverage information")
        parser.add_option("--cover-xml-file", action="store",
                          default=env.get('NOSE_COVER_XML_FILE',
```

```
                                        'coverage.xml'),
                        dest="cover_xml_file",
                        metavar="FILE",
                        help="Produce XML coverage information in file")
        parser.add_option("--cover-config-file", action="store",
                        default=env.get('NOSE_COVER_CONFIG_FILE'),
                        dest="cover_config_file",
                        help="Location of coverage config file "
                        "[NOSE_COVER_CONFIG_FILE]")
        parser.add_option("--cover-no-print", action="store_true",
                        default=env.get('NOSE_COVER_NO_PRINT'),
                        dest="cover_no_print",
                        help="Suppress printing of coverage information")

    def configure(self, options, conf):
        """
        Configure plugin.
        """
        try:
            self.status.pop('active')
        except KeyError:
            pass
        super(Coverage, self).configure(options, conf)
        if self.enabled:
            try:
                import coverage
                if not hasattr(coverage, 'coverage'):
                    raise ImportError("Unable to import coverage module")
            except ImportError:
                log.error("Coverage not available: "
                          "unable to import coverage module")
                self.enabled = False
                return
        self.conf = conf
        self.coverErase = bool(options.cover_erase)
        self.coverTests = bool(options.cover_tests)
        self.coverPackages = []
        if options.cover_packages:
            if isinstance(options.cover_packages, (list, tuple)):
                cover_packages = options.cover_packages
            else:
                cover_packages = [options.cover_packages]
            for pkgs in [tolist(x) for x in cover_packages]:
                self.coverPackages.extend(pkgs)
        self.coverInclusive = options.cover_inclusive
        if self.coverPackages:
            log.info("Coverage report will include only packages: %s",
                     self.coverPackages)
        self.coverHtmlDir = None
        if options.cover_html:
            self.coverHtmlDir = options.cover_html_dir
            log.debug('Will put HTML coverage report in %s', self.coverHtmlDir)
        self.coverBranches = options.cover_branches
        self.coverXmlFile = None
        if options.cover_min_percentage:
            self.coverMinPercentage = int(options.cover_min_percentage.rstrip('%'))
        if options.cover_xml:
            self.coverXmlFile = options.cover_xml_file
```

```
            log.debug('Will put XML coverage report in %s', self.coverXmlFile)
        # Coverage uses True to mean default
        self.coverConfigFile = True
        if options.cover_config_file:
            self.coverConfigFile = options.cover_config_file
        self.coverPrint = not options.cover_no_print
        if self.enabled:
            self.status['active'] = True
            self.coverInstance = coverage.coverage(auto_data=False,
                branch=self.coverBranches, data_suffix=conf.worker,
                source=self.coverPackages, config_file=self.coverConfigFile)
            self.coverInstance._warn_no_data = False
            self.coverInstance.is_worker = conf.worker
            self.coverInstance.exclude('#pragma[: ]+[nN][oO] [cC][oO][vV][eE][rR]')

            log.debug("Coverage begin")
            self.skipModules = sys.modules.keys()[:]
            if self.coverErase:
                log.debug("Clearing previously collected coverage statistics")
                self.coverInstance.combine()
                self.coverInstance.erase()

            if not self.coverInstance.is_worker:
                self.coverInstance.load()
                self.coverInstance.start()

    def beforeTest(self, *args, **kwargs):
        """
        Begin recording coverage information.
        """

        if self.coverInstance.is_worker:
            self.coverInstance.load()
            self.coverInstance.start()

    def afterTest(self, *args, **kwargs):
        """
        Stop recording coverage information.
        """

        if self.coverInstance.is_worker:
            self.coverInstance.stop()
            self.coverInstance.save()

    def report(self, stream):
        """
        Output code coverage report.
        """
        log.debug("Coverage report")
        self.coverInstance.stop()
        self.coverInstance.combine()
        self.coverInstance.save()
        modules = [module
                   for name, module in sys.modules.items()
                   if self.wantModuleCoverage(name, module)]
        log.debug("Coverage report will cover modules: %s", modules)
        if self.coverPrint:
            self.coverInstance.report(modules, file=stream)
```

```
        import coverage
        if self.coverHtmlDir:
            log.debug("Generating HTML coverage report")
            try:
                self.coverInstance.html_report(modules, self.coverHtmlDir)
            except coverage.misc.CoverageException, e:
                log.warning("Failed to generate HTML report: %s" % str(e))

        if self.coverXmlFile:
            log.debug("Generating XML coverage report")
            try:
                self.coverInstance.xml_report(modules, self.coverXmlFile)
            except coverage.misc.CoverageException, e:
                log.warning("Failed to generate XML report: %s" % str(e))

        # make sure we have minimum required coverage
        if self.coverMinPercentage:
            f = StringIO.StringIO()
            self.coverInstance.report(modules, file=f)

            multiPackageRe = (r'-------\s\w+\s+\d+\s+\d+(?:\s+\d+\s+\d+)?'
                              r'\s+(\d+)%\s+\d*\s{0,1}$')
            singlePackageRe = (r'-------\s[\w./]+\s+\d+\s+\d+(?:\s+\d+\s+\d+)?'
                               r'\s+(\d+)%(?:\s+[-\d, ]+)\s{0,1}$')

            m = re.search(multiPackageRe, f.getvalue())
            if m is None:
                m = re.search(singlePackageRe, f.getvalue())

            if m:
                percentage = int(m.groups()[0])
                if percentage < self.coverMinPercentage:
                    log.error('TOTAL Coverage did not reach minimum '
                              'required: %d%%' % self.coverMinPercentage)
                    sys.exit(1)
            else:
                log.error("No total percentage was found in coverage output, "
                          "something went wrong.")

    def wantModuleCoverage(self, name, module):
        if not hasattr(module, '__file__'):
            log.debug("no coverage of %s: no __file__", name)
            return False
        module_file = src(module.__file__)
        if not module_file or not module_file.endswith('.py'):
            log.debug("no coverage of %s: not a python file", name)
            return False
        if self.coverPackages:
            for package in self.coverPackages:
                if (re.findall(r'^%s\b' % re.escape(package), name)
                    and (self.coverTests
                         or not self.conf.testMatch.search(name))):
                    log.debug("coverage for %s", name)
                    return True
        if name in self.skipModules:
            log.debug("no coverage for %s: loaded before coverage start",
                      name)
```

```
            return False
        if self.conf.testMatch.search(name) and not self.coverTests:
            log.debug("no coverage for %s: is a test", name)
            return False
        # accept any package that passed the previous tests, unless
        # coverPackages is on -- in that case, if we wanted this
        # module, we would have already returned True
        return not self.coverPackages

    def wantFile(self, file, package=None):
        """If inclusive coverage enabled, return true for all source files
        in wanted packages.
        """
        if self.coverInclusive:
            if file.endswith(".py"):
                if package and self.coverPackages:
                    for want in self.coverPackages:
                        if package.startswith(want):
                            return True
                else:
                    return True
        return None
```

Debug: drop into pdb on errors or failures

This plugin provides `--pdb` and `--pdb-failures` options. The `--pdb` option will drop the test runner into pdb when it encounters an error. To drop into pdb on failure, use `--pdb-failures`.

Options

`--pdb`
Drop into debugger on failures or errors

`--pdb-failures`
Drop into debugger on failures

`--pdb-errors`
Drop into debugger on errors

Plugin

class `nose.plugins.debug.`**`Pdb`**
Bases: *nose.plugins.base.Plugin*

Provides –pdb and –pdb-failures options that cause the test runner to drop into pdb if it encounters an error or failure, respectively.

`addError`(*test*, *err*)
Enter pdb if configured to debug errors.

`addFailure`(*test*, *err*)
Enter pdb if configured to debug failures.

`configure`(*options*, *conf*)
Configure which kinds of exceptions trigger plugin.

options (*parser*, *env*)
Register commandline options.

Source

```
"""
This plugin provides ``--pdb`` and ``--pdb-failures`` options. The ``--pdb``
option will drop the test runner into pdb when it encounters an error. To
drop into pdb on failure, use ``--pdb-failures``.
"""

import pdb
from nose.plugins.base import Plugin

class Pdb(Plugin):
    """
    Provides --pdb and --pdb-failures options that cause the test runner to
    drop into pdb if it encounters an error or failure, respectively.
    """
    enabled_for_errors = False
    enabled_for_failures = False
    score = 5 # run last, among builtins

    def options(self, parser, env):
        """Register commandline options.
        """
        parser.add_option(
            "--pdb", action="store_true", dest="debugBoth",
            default=env.get('NOSE_PDB', False),
            help="Drop into debugger on failures or errors")
        parser.add_option(
            "--pdb-failures", action="store_true",
            dest="debugFailures",
            default=env.get('NOSE_PDB_FAILURES', False),
            help="Drop into debugger on failures")
        parser.add_option(
            "--pdb-errors", action="store_true",
            dest="debugErrors",
            default=env.get('NOSE_PDB_ERRORS', False),
            help="Drop into debugger on errors")

    def configure(self, options, conf):
        """Configure which kinds of exceptions trigger plugin.
        """
        self.conf = conf
        self.enabled_for_errors = options.debugErrors or options.debugBoth
        self.enabled_for_failures = options.debugFailures or options.debugBoth
        self.enabled = self.enabled_for_failures or self.enabled_for_errors

    def addError(self, test, err):
        """Enter pdb if configured to debug errors.
        """
        if not self.enabled_for_errors:
            return
        self.debug(err)

    def addFailure(self, test, err):
```

```
        """Enter pdb if configured to debug failures.
        """
        if not self.enabled_for_failures:
            return
        self.debug(err)

    def debug(self, err):
        import sys # FIXME why is this import here?
        ec, ev, tb = err
        stdout = sys.stdout
        sys.stdout = sys.__stdout__
        try:
            pdb.post_mortem(tb)
        finally:
            sys.stdout = stdout
```

Deprecated: mark tests as deprecated

This plugin installs a DEPRECATED error class for the `DeprecatedTest` exception. When `DeprecatedTest` is raised, the exception will be logged in the deprecated attribute of the result, `D` or `DEPRECATED` (verbose) will be output, and the exception will not be counted as an error or failure. It is enabled by default, but can be turned off by using `--no-deprecated`.

Options

--no-deprecated
Disable special handling of DeprecatedTest exceptions.

Plugin

class nose.plugins.deprecated.**Deprecated**
Bases: *nose.plugins.errorclass.ErrorClassPlugin*

Installs a DEPRECATED error class for the DeprecatedTest exception. Enabled by default.

configure(*options*, *conf*)
Configure plugin.

options(*parser*, *env*)
Register commandline options.

Source

```
"""
This plugin installs a DEPRECATED error class for the :class:`DeprecatedTest`
exception. When :class:`DeprecatedTest` is raised, the exception will be logged
in the deprecated attribute of the result, ``D`` or ``DEPRECATED`` (verbose)
will be output, and the exception will not be counted as an error or failure.
It is enabled by default, but can be turned off by using ``--no-deprecated``.
"""

from nose.plugins.errorclass import ErrorClass, ErrorClassPlugin
```

```
class DeprecatedTest(Exception):
    """Raise this exception to mark a test as deprecated.
    """
    pass


class Deprecated(ErrorClassPlugin):
    """
    Installs a DEPRECATED error class for the DeprecatedTest exception. Enabled
    by default.
    """
    enabled = True
    deprecated = ErrorClass(DeprecatedTest,
                            label='DEPRECATED',
                            isfailure=False)

    def options(self, parser, env):
        """Register commandline options.
        """
        env_opt = 'NOSE_WITHOUT_DEPRECATED'
        parser.add_option('--no-deprecated', action='store_true',
                          dest='noDeprecated', default=env.get(env_opt, False),
                          help="Disable special handling of DeprecatedTest "
                          "exceptions.")

    def configure(self, options, conf):
        """Configure plugin.
        """
        if not self.can_configure:
            return
        self.conf = conf
        disable = getattr(options, 'noDeprecated', False)
        if disable:
            self.enabled = False
```

Doctests: run doctests with nose

Use the Doctest plugin with `--with-doctest` or the NOSE_WITH_DOCTEST environment variable to enable collection and execution of `doctests`. Because doctests are usually included in the tested package (instead of being grouped into packages or modules of their own), nose only looks for them in the non-test packages it discovers in the working directory.

Doctests may also be placed into files other than python modules, in which case they can be collected and executed by using the `--doctest-extension` switch or NOSE_DOCTEST_EXTENSION environment variable to indicate which file extension(s) to load.

When loading doctests from non-module files, use the `--doctest-fixtures` switch to specify how to find modules containing fixtures for the tests. A module name will be produced by appending the value of that switch to the base name of each doctest file loaded. For example, a doctest file "widgets.rst" with the switch `--doctest_fixtures=_fixt` will load fixtures from the module `widgets_fixt.py`.

A fixtures module may define any or all of the following functions:

- setup([module]) or setup_module([module])

Called before the test runs. You may raise SkipTest to skip all tests.

- teardown([module]) or teardown_module([module])

  Called after the test runs, if setup/setup_module did not raise an unhandled exception.

- setup_test(test)

  Called before the test. NOTE: the argument passed is a doctest.DocTest instance, *not* a unittest.TestCase.

- teardown_test(test)

  Called after the test, if setup_test did not raise an exception. NOTE: the argument passed is a doctest.DocTest instance, *not* a unittest.TestCase.

Doctests are run like any other test, with the exception that output capture does not work; doctest does its own output capture while running a test.

Note: See *Doctest Fixtures* for additional documentation and examples.

Options

--with-doctest
Enable plugin Doctest: Activate doctest plugin to find and run doctests in non-test modules. [NOSE_WITH_DOCTEST]

--doctest-tests
Also look for doctests in test modules. Note that classes, methods and functions should have either doctests or non-doctest tests, not both. [NOSE_DOCTEST_TESTS]

--doctest-extension=EXT
Also look for doctests in files with this extension [NOSE_DOCTEST_EXTENSION]

--doctest-result-variable=VAR
Change the variable name set to the result of the last interpreter command from the default '_'. Can be used to avoid conflicts with the _() function used for text translation. [NOSE_DOCTEST_RESULT_VAR]

--doctest-fixtures=SUFFIX
Find fixtures for a doctest file in module with this name appended to the base name of the doctest file

--doctest-options=OPTIONS
Specify options to pass to doctest. Eg. '+ELLIPSIS,+NORMALIZE_WHITESPACE'

Plugin

class nose.plugins.doctests.**Doctest**
Bases: *nose.plugins.base.Plugin*

Activate doctest plugin to find and run doctests in non-test modules.

configure(*options*, *config*)
Configure plugin.

loadTestsFromFile(*filename*)
Load doctests from the file.

Tests are loaded only if filename's extension matches configured doctest extension.

loadTestsFromModule (*module*)
Load doctests from the module.

makeTest (*obj*, *parent*)
Look for doctests in the given object, which will be a function, method or class.

options (*parser*, *env*)
Register commmandline options.

prepareTestLoader (*loader*)
Capture loader's suiteClass.

This is used to create test suites from doctest files.

suiteClass
alias of `DoctestSuite`

wantFile (*file*)
Override to select all modules and any file ending with configured doctest extension.

Source

```
"""Use the Doctest plugin with ``--with-doctest`` or the NOSE_WITH_DOCTEST
environment variable to enable collection and execution of :mod:`doctests
<doctest>`.  Because doctests are usually included in the tested package
(instead of being grouped into packages or modules of their own), nose only
looks for them in the non-test packages it discovers in the working directory.

Doctests may also be placed into files other than python modules, in which
case they can be collected and executed by using the ``--doctest-extension``
switch or NOSE_DOCTEST_EXTENSION environment variable to indicate which file
extension(s) to load.

When loading doctests from non-module files, use the ``--doctest-fixtures``
switch to specify how to find modules containing fixtures for the tests. A
module name will be produced by appending the value of that switch to the base
name of each doctest file loaded. For example, a doctest file "widgets.rst"
with the switch ``--doctest_fixtures=_fixt`` will load fixtures from the module
``widgets_fixt.py``.

A fixtures module may define any or all of the following functions:

* setup([module]) or setup_module([module])

  Called before the test runs. You may raise SkipTest to skip all tests.

* teardown([module]) or teardown_module([module])

  Called after the test runs, if setup/setup_module did not raise an
  unhandled exception.

* setup_test(test)

  Called before the test. NOTE: the argument passed is a
  doctest.DocTest instance, *not* a unittest.TestCase.

* teardown_test(test)
```

```
  Called after the test, if setup_test did not raise an exception. NOTE: the
  argument passed is a doctest.DocTest instance, *not* a unittest.TestCase.

Doctests are run like any other test, with the exception that output
capture does not work; doctest does its own output capture while running a
test.

.. note ::

   See :doc:`../doc_tests/test_doctest_fixtures/doctest_fixtures` for
   additional documentation and examples.

"""
from __future__ import generators

import logging
import os
import sys
import unittest
from inspect import getmodule
from nose.plugins.base import Plugin
from nose.suite import ContextList
from nose.util import anyp, getpackage, test_address, resolve_name, \
     src, tolist, isproperty
try:
    from cStringIO import StringIO
except ImportError:
    from StringIO import StringIO
import sys
import __builtin__ as builtin_mod

log = logging.getLogger(__name__)

try:
    import doctest
    doctest.DocTestCase
    # system version of doctest is acceptable, but needs a monkeypatch
except (ImportError, AttributeError):
    # system version is too old
    import nose.ext.dtcompat as doctest


#
# Doctest and coverage don't get along, so we need to create
# a monkeypatch that will replace the part of doctest that
# interferes with coverage reports.
#
# The monkeypatch is based on this zope patch:
# http://svn.zope.org/Zope3/trunk/src/zope/testing/doctest.py?rev=28679&r1=28703&
↪r2=28705
#
_orp = doctest._OutputRedirectingPdb

class NoseOutputRedirectingPdb(_orp):
    def __init__(self, out):
        self.__debugger_used = False
        _orp.__init__(self, out)
```

```
    def set_trace(self):
        self.__debugger_used = True
        _orp.set_trace(self, sys._getframe().f_back)

    def set_continue(self):
        # Calling set_continue unconditionally would break unit test
        # coverage reporting, as Bdb.set_continue calls sys.settrace(None).
        if self.__debugger_used:
            _orp.set_continue(self)
doctest._OutputRedirectingPdb = NoseOutputRedirectingPdb


class DoctestSuite(unittest.TestSuite):
    """
    Doctest suites are parallelizable at the module or file level only,
    since they may be attached to objects that are not individually
    addressable (like properties). This suite subclass is used when
    loading doctests from a module to ensure that behavior.

    This class is used only if the plugin is not fully prepared;
    in normal use, the loader's suiteClass is used.

    """
    can_split = False

    def __init__(self, tests=(), context=None, can_split=False):
        self.context = context
        self.can_split = can_split
        unittest.TestSuite.__init__(self, tests=tests)

    def address(self):
        return test_address(self.context)

    def __iter__(self):
        # 2.3 compat
        return iter(self._tests)

    def __str__(self):
        return str(self._tests)


class Doctest(Plugin):
    """
    Activate doctest plugin to find and run doctests in non-test modules.
    """
    extension = None
    suiteClass = DoctestSuite

    def options(self, parser, env):
        """Register commmandline options.
        """
        Plugin.options(self, parser, env)
        parser.add_option('--doctest-tests', action='store_true',
                          dest='doctest_tests',
                          default=env.get('NOSE_DOCTEST_TESTS'),
                          help="Also look for doctests in test modules. "
                          "Note that classes, methods and functions should "
                          "have either doctests or non-doctest tests, "
```

```
                        "not both. [NOSE_DOCTEST_TESTS]")
        parser.add_option('--doctest-extension', action="append",
                          dest="doctestExtension",
                          metavar="EXT",
                          help="Also look for doctests in files with "
                          "this extension [NOSE_DOCTEST_EXTENSION]")
        parser.add_option('--doctest-result-variable',
                          dest='doctest_result_var',
                          default=env.get('NOSE_DOCTEST_RESULT_VAR'),
                          metavar="VAR",
                          help="Change the variable name set to the result of "
                          "the last interpreter command from the default '_'. "
                          "Can be used to avoid conflicts with the _() "
                          "function used for text translation. "
                          "[NOSE_DOCTEST_RESULT_VAR]")
        parser.add_option('--doctest-fixtures', action="store",
                          dest="doctestFixtures",
                          metavar="SUFFIX",
                          help="Find fixtures for a doctest file in module "
                          "with this name appended to the base name "
                          "of the doctest file")
        parser.add_option('--doctest-options', action="append",
                          dest="doctestOptions",
                          metavar="OPTIONS",
                          help="Specify options to pass to doctest. " +
                          "Eg. '+ELLIPSIS,+NORMALIZE_WHITESPACE'")
        # Set the default as a list, if given in env; otherwise
        # an additional value set on the command line will cause
        # an error.
        env_setting = env.get('NOSE_DOCTEST_EXTENSION')
        if env_setting is not None:
            parser.set_defaults(doctestExtension=tolist(env_setting))

    def configure(self, options, config):
        """Configure plugin.
        """
        Plugin.configure(self, options, config)
        self.doctest_result_var = options.doctest_result_var
        self.doctest_tests = options.doctest_tests
        self.extension = tolist(options.doctestExtension)
        self.fixtures = options.doctestFixtures
        self.finder = doctest.DocTestFinder()
        self.optionflags = 0
        if options.doctestOptions:
            flags = ",".join(options.doctestOptions).split(',')
            for flag in flags:
                if not flag or flag[0] not in '+-':
                    raise ValueError(
                        "Must specify doctest options with starting " +
                        "'+' or '-'.  Got %s" % (flag,))
                mode, option_name = flag[0], flag[1:]
                option_flag = doctest.OPTIONFLAGS_BY_NAME.get(option_name)
                if not option_flag:
                    raise ValueError("Unknown doctest option %s" %
                                     (option_name,))
                if mode == '+':
                    self.optionflags |= option_flag
                elif mode == '-':
```

```
                    self.optionflags &= ~option_flag

    def prepareTestLoader(self, loader):
        """Capture loader's suiteClass.

        This is used to create test suites from doctest files.

        """
        self.suiteClass = loader.suiteClass

    def loadTestsFromModule(self, module):
        """Load doctests from the module.
        """
        log.debug("loading from %s", module)
        if not self.matches(module.__name__):
            log.debug("Doctest doesn't want module %s", module)
            return
        try:
            tests = self.finder.find(module)
        except AttributeError:
            log.exception("Attribute error loading from %s", module)
            # nose allows module.__test__ = False; doctest does not and throws
            # AttributeError
            return
        if not tests:
            log.debug("No tests found in %s", module)
            return
        tests.sort()
        module_file = src(module.__file__)
        # FIXME this breaks the id plugin somehow (tests probably don't
        # get wrapped in result proxy or something)
        cases = []
        for test in tests:
            if not test.examples:
                continue
            if not test.filename:
                test.filename = module_file
            cases.append(DocTestCase(test,
                                     optionflags=self.optionflags,
                                     result_var=self.doctest_result_var))
        if cases:
            yield self.suiteClass(cases, context=module, can_split=False)

    def loadTestsFromFile(self, filename):
        """Load doctests from the file.

        Tests are loaded only if filename's extension matches
        configured doctest extension.

        """
        if self.extension and anyp(filename.endswith, self.extension):
            name = os.path.basename(filename)
            dh = open(filename)
            try:
                doc = dh.read()
            finally:
                dh.close()
```

```
            fixture_context = None
            globs = {'__file__': filename}
            if self.fixtures:
                base, ext = os.path.splitext(name)
                dirname = os.path.dirname(filename)
                sys.path.append(dirname)
                fixt_mod = base + self.fixtures
                try:
                    fixture_context = __import__(
                        fixt_mod, globals(), locals(), ["nop"])
                except ImportError, e:
                    log.debug(
                        "Could not import %s: %s (%s)", fixt_mod, e, sys.path)
                log.debug("Fixture module %s resolved to %s",
                          fixt_mod, fixture_context)
                if hasattr(fixture_context, 'globs'):
                    globs = fixture_context.globs(globs)
            parser = doctest.DocTestParser()
            test = parser.get_doctest(
                doc, globs=globs, name=name,
                filename=filename, lineno=0)
            if test.examples:
                case = DocFileCase(
                    test,
                    optionflags=self.optionflags,
                    setUp=getattr(fixture_context, 'setup_test', None),
                    tearDown=getattr(fixture_context, 'teardown_test', None),
                    result_var=self.doctest_result_var)
                if fixture_context:
                    yield ContextList((case,), context=fixture_context)
                else:
                    yield case
            else:
                yield False # no tests to load

    def makeTest(self, obj, parent):
        """Look for doctests in the given object, which will be a
        function, method or class.
        """
        name = getattr(obj, '__name__', 'Unnammed %s' % type(obj))
        doctests = self.finder.find(obj, module=getmodule(parent), name=name)
        if doctests:
            for test in doctests:
                if len(test.examples) == 0:
                    continue
                yield DocTestCase(test, obj=obj, optionflags=self.optionflags,
                                  result_var=self.doctest_result_var)

    def matches(self, name):
        # FIXME this seems wrong -- nothing is ever going to
        # fail this test, since we're given a module NAME not FILE
        if name == '__init__.py':
            return False
        # FIXME don't think we need include/exclude checks here?
        return ((self.doctest_tests or not self.conf.testMatch.search(name)
                 or (self.conf.include
                     and filter(None,
                                [inc.search(name)
```

```
                                    for inc in self.conf.include])))
                and (not self.conf.exclude
                     or not filter(None,
                                   [exc.search(name)
                                    for exc in self.conf.exclude])))

    def wantFile(self, file):
        """Override to select all modules and any file ending with
        configured doctest extension.
        """
        # always want .py files
        if file.endswith('.py'):
            return True
        # also want files that match my extension
        if (self.extension
            and anyp(file.endswith, self.extension)
            and (not self.conf.exclude
                 or not filter(None,
                               [exc.search(file)
                                for exc in self.conf.exclude]))):
            return True
        return None


class DocTestCase(doctest.DocTestCase):
    """Overrides DocTestCase to
    provide an address() method that returns the correct address for
    the doctest case. To provide hints for address(), an obj may also
    be passed -- this will be used as the test object for purposes of
    determining the test address, if it is provided.
    """
    def __init__(self, test, optionflags=0, setUp=None, tearDown=None,
                 checker=None, obj=None, result_var='_'):
        self._result_var = result_var
        self._nose_obj = obj
        super(DocTestCase, self).__init__(
            test, optionflags=optionflags, setUp=setUp, tearDown=tearDown,
            checker=checker)

    def address(self):
        if self._nose_obj is not None:
            return test_address(self._nose_obj)
        obj = resolve_name(self._dt_test.name)

        if isproperty(obj):
            # properties have no connection to the class they are in
            # so we can't just look 'em up, we have to first look up
            # the class, then stick the prop on the end
            parts = self._dt_test.name.split('.')
            class_name = '.'.join(parts[:-1])
            cls = resolve_name(class_name)
            base_addr = test_address(cls)
            return (base_addr[0], base_addr[1],
                    '.'.join([base_addr[2], parts[-1]]))
        else:
            return test_address(obj)

    # doctests loaded via find(obj) omit the module name
```

```
    # so we need to override id, __repr__ and shortDescription
    # bonus: this will squash a 2.3 vs 2.4 incompatiblity
    def id(self):
        name = self._dt_test.name
        filename = self._dt_test.filename
        if filename is not None:
            pk = getpackage(filename)
            if pk is None:
                return name
            if not name.startswith(pk):
                name = "%s.%s" % (pk, name)
        return name

    def __repr__(self):
        name = self.id()
        name = name.split('.')
        return "%s (%s)" % (name[-1], '.'.join(name[:-1]))
    __str__ = __repr__

    def shortDescription(self):
        return 'Doctest: %s' % self.id()

    def setUp(self):
        if self._result_var is not None:
            self._old_displayhook = sys.displayhook
            sys.displayhook = self._displayhook
        super(DocTestCase, self).setUp()

    def _displayhook(self, value):
        if value is None:
            return
        setattr(builtin_mod, self._result_var,  value)
        print repr(value)

    def tearDown(self):
        super(DocTestCase, self).tearDown()
        if self._result_var is not None:
            sys.displayhook = self._old_displayhook
            delattr(builtin_mod, self._result_var)


class DocFileCase(doctest.DocFileCase):
    """Overrides to provide address() method that returns the correct
    address for the doc file case.
    """
    def __init__(self, test, optionflags=0, setUp=None, tearDown=None,
                 checker=None, result_var='_'):
        self._result_var = result_var
        super(DocFileCase, self).__init__(
            test, optionflags=optionflags, setUp=setUp, tearDown=tearDown,
            checker=None)

    def address(self):
        return (self._dt_test.filename, None, None)

    def setUp(self):
        if self._result_var is not None:
            self._old_displayhook = sys.displayhook
```

```
            sys.displayhook = self._displayhook
        super(DocFileCase, self).setUp()

    def _displayhook(self, value):
        if value is None:
            return
        setattr(builtin_mod, self._result_var, value)
        print repr(value)

    def tearDown(self):
        super(DocFileCase, self).tearDown()
        if self._result_var is not None:
            sys.displayhook = self._old_displayhook
            delattr(builtin_mod, self._result_var)
```

Failure Detail: introspect asserts

This plugin provides assert introspection. When the plugin is enabled and a test failure occurs, the traceback is displayed with extra context around the line in which the exception was raised. Simple variable substitution is also performed in the context output to provide more debugging information.

Options

-d, --detailed-errors, --failure-detail
Add detail to error output by attempting to evaluate failed asserts [NOSE_DETAILED_ERRORS]

Plugin

class nose.plugins.failuredetail.**FailureDetail**
Bases: *nose.plugins.base.Plugin*

Plugin that provides extra information in tracebacks of test failures.

configure(*options*, *conf*)
Configure plugin.

formatFailure(*test*, *err*)
Add detail from traceback inspection to error message of a failure.

options(*parser*, *env*)
Register commmandline options.

Source

```
"""
This plugin provides assert introspection. When the plugin is enabled
and a test failure occurs, the traceback is displayed with extra context
around the line in which the exception was raised. Simple variable
substitution is also performed in the context output to provide more
debugging information.
"""
```

```
from nose.plugins import Plugin
from nose.pyversion import exc_to_unicode, force_unicode
from nose.inspector import inspect_traceback

class FailureDetail(Plugin):
    """
    Plugin that provides extra information in tracebacks of test failures.
    """
    score = 1600 # before capture

    def options(self, parser, env):
        """Register commmandline options.
        """
        parser.add_option(
            "-d", "--detailed-errors", "--failure-detail",
            action="store_true",
            default=env.get('NOSE_DETAILED_ERRORS'),
            dest="detailedErrors", help="Add detail to error"
            " output by attempting to evaluate failed"
            " asserts [NOSE_DETAILED_ERRORS]")

    def configure(self, options, conf):
        """Configure plugin.
        """
        if not self.can_configure:
            return
        self.enabled = options.detailedErrors
        self.conf = conf

    def formatFailure(self, test, err):
        """Add detail from traceback inspection to error message of a failure.
        """
        ec, ev, tb = err
        tbinfo, str_ev = None, exc_to_unicode(ev)

        if tb:
            tbinfo = force_unicode(inspect_traceback(tb))
            str_ev = '\n'.join([str_ev, tbinfo])
        test.tbinfo = tbinfo
        return (ec, str_ev, tb)
```

Isolate: protect tests from (some) side-effects

The isolation plugin resets the contents of sys.modules after running each test module or package. Use it by setting `--with-isolation` or the NOSE_WITH_ISOLATION environment variable.

The effects are similar to wrapping the following functions around the import and execution of each test module:

```
def setup(module):
    module._mods = sys.modules.copy()

def teardown(module):
    to_del = [ m for m in sys.modules.keys() if m not in
               module._mods ]
    for mod in to_del:
```

```
        del sys.modules[mod]
    sys.modules.update(module._mods)
```

Isolation works only during lazy loading. In normal use, this is only during discovery of modules within a directory, where the process of importing, loading tests and running tests from each module is encapsulated in a single loadTestsFromName call. This plugin implements loadTestsFromNames to force the same lazy-loading there, which allows isolation to work in directed mode as well as discovery, at the cost of some efficiency: lazy-loading names forces full context setup and teardown to run for each name, defeating the grouping that is normally used to ensure that context setup and teardown are run the fewest possible times for a given set of names.

Warning: This plugin should not be used in conjunction with other plugins that assume that modules, once imported, will stay imported; for instance, it may cause very odd results when used with the coverage plugin.

Options

--with-isolation
Enable plugin IsolationPlugin: Activate the isolation plugin to isolate changes to external modules to a single test module or package. The isolation plugin resets the contents of sys.modules after each test module or package runs to its state before the test. PLEASE NOTE that this plugin should not be used with the coverage plugin, or in any other case where module reloading may produce undesirable side-effects. [NOSE_WITH_ISOLATION]

Plugin

class nose.plugins.isolate.**IsolationPlugin**
Bases: *nose.plugins.base.Plugin*

Activate the isolation plugin to isolate changes to external modules to a single test module or package. The isolation plugin resets the contents of sys.modules after each test module or package runs to its state before the test. PLEASE NOTE that this plugin should not be used with the coverage plugin, or in any other case where module reloading may produce undesirable side-effects.

afterContext()
Pop my mod stack and restore sys.modules to the state it was in when mod stack was pushed.

beforeContext()
Copy sys.modules onto my mod stack

configure(*options*, *conf*)
Configure plugin.

loadTestsFromNames(*names*, *module=None*)
Create a lazy suite that calls beforeContext and afterContext around each name. The side-effect of this is that full context fixtures will be set up and torn down around each test named.

prepareTestLoader(*loader*)
Get handle on test loader so we can use it in loadTestsFromNames.

Source

```
"""The isolation plugin resets the contents of sys.modules after running
each test module or package. Use it by setting ``--with-isolation`` or the
NOSE_WITH_ISOLATION environment variable.
```

```
The effects are similar to wrapping the following functions around the
import and execution of each test module::

    def setup(module):
        module._mods = sys.modules.copy()

    def teardown(module):
        to_del = [ m for m in sys.modules.keys() if m not in
                   module._mods ]
        for mod in to_del:
            del sys.modules[mod]
        sys.modules.update(module._mods)

Isolation works only during lazy loading. In normal use, this is only
during discovery of modules within a directory, where the process of
importing, loading tests and running tests from each module is
encapsulated in a single loadTestsFromName call. This plugin
implements loadTestsFromNames to force the same lazy-loading there,
which allows isolation to work in directed mode as well as discovery,
at the cost of some efficiency: lazy-loading names forces full context
setup and teardown to run for each name, defeating the grouping that
is normally used to ensure that context setup and teardown are run the
fewest possible times for a given set of names.

.. warning ::

    This plugin should not be used in conjunction with other plugins
    that assume that modules, once imported, will stay imported; for
    instance, it may cause very odd results when used with the coverage
    plugin.

"""

import logging
import sys

from nose.plugins import Plugin


log = logging.getLogger('nose.plugins.isolation')

class IsolationPlugin(Plugin):
    """
    Activate the isolation plugin to isolate changes to external
    modules to a single test module or package. The isolation plugin
    resets the contents of sys.modules after each test module or
    package runs to its state before the test. PLEASE NOTE that this
    plugin should not be used with the coverage plugin, or in any other case
    where module reloading may produce undesirable side-effects.
    """
    score = 10 # I want to be last
    name = 'isolation'

    def configure(self, options, conf):
        """Configure plugin.
        """
        Plugin.configure(self, options, conf)
```

```
        self._mod_stack = []

    def beforeContext(self):
        """Copy sys.modules onto my mod stack
        """
        mods = sys.modules.copy()
        self._mod_stack.append(mods)

    def afterContext(self):
        """Pop my mod stack and restore sys.modules to the state
        it was in when mod stack was pushed.
        """
        mods = self._mod_stack.pop()
        to_del = [ m for m in sys.modules.keys() if m not in mods ]
        if to_del:
            log.debug('removing sys modules entries: %s', to_del)
            for mod in to_del:
                del sys.modules[mod]
        sys.modules.update(mods)

    def loadTestsFromNames(self, names, module=None):
        """Create a lazy suite that calls beforeContext and afterContext
        around each name. The side-effect of this is that full context
        fixtures will be set up and torn down around each test named.
        """
        # Fast path for when we don't care
        if not names or len(names) == 1:
            return
        loader = self.loader
        plugins = self.conf.plugins
        def lazy():
            for name in names:
                plugins.beforeContext()
                yield loader.loadTestsFromName(name, module=module)
                plugins.afterContext()
        return (loader.suiteClass(lazy), [])

    def prepareTestLoader(self, loader):
        """Get handle on test loader so we can use it in loadTestsFromNames.
        """
        self.loader = loader
```

Logcapture: capture logging during tests

This plugin captures logging statements issued during test execution. When an error or failure occurs, the captured log messages are attached to the running test in the test.capturedLogging attribute, and displayed with the error failure output. It is enabled by default but can be turned off with the option `--nologcapture`.

You can filter captured logging statements with the `--logging-filter` option. If set, it specifies which logger(s) will be captured; loggers that do not match will be passed. Example: specifying `--logging-filter=sqlalchemy,myapp` will ensure that only statements logged via sqlalchemy.engine, myapp or myapp.foo.bar logger will be logged.

You can remove other installed logging handlers with the `--logging-clear-handlers` option.

Options

--nologcapture
Disable logging capture plugin. Logging configuration will be left intact. [NOSE_NOLOGCAPTURE]

--logging-format=FORMAT
Specify custom format to print statements. Uses the same format as used by standard logging handlers. [NOSE_LOGFORMAT]

--logging-datefmt=FORMAT
Specify custom date/time format to print statements. Uses the same format as used by standard logging handlers. [NOSE_LOGDATEFMT]

--logging-filter=FILTER
Specify which statements to filter in/out. By default, everything is captured. If the output is too verbose, use this option to filter out needless output. Example: filter=foo will capture statements issued ONLY to foo or foo.what.ever.sub but not foobar or other logger. Specify multiple loggers with comma: filter=foo,bar,baz. If any logger name is prefixed with a minus, eg filter=-foo, it will be excluded rather than included. Default: exclude logging messages from nose itself (-nose). [NOSE_LOGFILTER]

--logging-clear-handlers
Clear all other logging handlers

--logging-level=DEFAULT
Set the log level to capture

Plugin

class nose.plugins.logcapture.**LogCapture**
Bases: *nose.plugins.base.Plugin*

Log capture plugin. Enabled by default. Disable with –nologcapture. This plugin captures logging statements issued during test execution, appending any output captured to the error or failure output, should the test fail or raise an error.

afterTest(*test*)
Clear buffers after test.

beforeTest(*test*)
Clear buffers and handlers before test.

begin()
Set up logging handler before test run begins.

configure(*options*, *conf*)
Configure plugin.

formatError(*test*, *err*)
Add captured log messages to error output.

formatFailure(*test*, *err*)
Add captured log messages to failure output.

options(*parser*, *env*)
Register commandline options.

Source

```
"""
This plugin captures logging statements issued during test execution. When an
error or failure occurs, the captured log messages are attached to the running
test in the test.capturedLogging attribute, and displayed with the error failure
output. It is enabled by default but can be turned off with the option
``--nologcapture``.

You can filter captured logging statements with the ``--logging-filter`` option.
If set, it specifies which logger(s) will be captured; loggers that do not match
will be passed. Example: specifying ``--logging-filter=sqlalchemy,myapp``
will ensure that only statements logged via sqlalchemy.engine, myapp
or myapp.foo.bar logger will be logged.

You can remove other installed logging handlers with the
``--logging-clear-handlers`` option.
"""

import logging
from logging import Handler
import threading

from nose.plugins.base import Plugin
from nose.util import anyp, ln, safe_str

try:
    from cStringIO import StringIO
except ImportError:
    from StringIO import StringIO

log = logging.getLogger(__name__)

class FilterSet(object):
    def __init__(self, filter_components):
        self.inclusive, self.exclusive = self._partition(filter_components)

    # @staticmethod
    def _partition(components):
        inclusive, exclusive = [], []
        for component in components:
            if component.startswith('-'):
                exclusive.append(component[1:])
            else:
                inclusive.append(component)
        return inclusive, exclusive
    _partition = staticmethod(_partition)

    def allow(self, record):
        """returns whether this record should be printed"""
        if not self:
            # nothing to filter
            return True
        return self._allow(record) and not self._deny(record)

    # @staticmethod
    def _any_match(matchers, record):
        """return the bool of whether `record` starts with
```

```
        any item in `matchers`"""
        def record_matches_key(key):
            return record == key or record.startswith(key + '.')
        return anyp(bool, map(record_matches_key, matchers))
    _any_match = staticmethod(_any_match)

    def _allow(self, record):
        if not self.inclusive:
            return True
        return self._any_match(self.inclusive, record)

    def _deny(self, record):
        if not self.exclusive:
            return False
        return self._any_match(self.exclusive, record)


class MyMemoryHandler(Handler):
    def __init__(self, logformat, logdatefmt, filters):
        Handler.__init__(self)
        fmt = logging.Formatter(logformat, logdatefmt)
        self.setFormatter(fmt)
        self.filterset = FilterSet(filters)
        self.buffer = []
    def emit(self, record):
        self.buffer.append(self.format(record))
    def flush(self):
        pass # do nothing
    def truncate(self):
        self.buffer = []
    def filter(self, record):
        if self.filterset.allow(record.name):
            return Handler.filter(self, record)
    def __getstate__(self):
        state = self.__dict__.copy()
        del state['lock']
        return state
    def __setstate__(self, state):
        self.__dict__.update(state)
        self.lock = threading.RLock()


class LogCapture(Plugin):
    """
    Log capture plugin. Enabled by default. Disable with --nologcapture.
    This plugin captures logging statements issued during test execution,
    appending any output captured to the error or failure output,
    should the test fail or raise an error.
    """
    enabled = True
    env_opt = 'NOSE_NOLOGCAPTURE'
    name = 'logcapture'
    score = 500
    logformat = '%(name)s: %(levelname)s: %(message)s'
    logdatefmt = None
    clear = False
    filters = ['-nose']
```

```
    def options(self, parser, env):
        """Register commandline options.
        """
        parser.add_option(
            "--nologcapture", action="store_false",
            default=not env.get(self.env_opt), dest="logcapture",
            help="Disable logging capture plugin. "
                 "Logging configuration will be left intact."
                 " [NOSE_NOLOGCAPTURE]")
        parser.add_option(
            "--logging-format", action="store", dest="logcapture_format",
            default=env.get('NOSE_LOGFORMAT') or self.logformat,
            metavar="FORMAT",
            help="Specify custom format to print statements. "
                 "Uses the same format as used by standard logging handlers."
                 " [NOSE_LOGFORMAT]")
        parser.add_option(
            "--logging-datefmt", action="store", dest="logcapture_datefmt",
            default=env.get('NOSE_LOGDATEFMT') or self.logdatefmt,
            metavar="FORMAT",
            help="Specify custom date/time format to print statements. "
                 "Uses the same format as used by standard logging handlers."
                 " [NOSE_LOGDATEFMT]")
        parser.add_option(
            "--logging-filter", action="store", dest="logcapture_filters",
            default=env.get('NOSE_LOGFILTER'),
            metavar="FILTER",
            help="Specify which statements to filter in/out. "
                 "By default, everything is captured. If the output is too"
                 " verbose,\nuse this option to filter out needless output.\n"
                 "Example: filter=foo will capture statements issued ONLY to\n"
                 " foo or foo.what.ever.sub but not foobar or other logger.\n"
                 "Specify multiple loggers with comma: filter=foo,bar,baz.\n"
                 "If any logger name is prefixed with a minus, eg filter=-foo,\n"
                 "it will be excluded rather than included. Default: "
                 "exclude logging messages from nose itself (-nose)."
                 " [NOSE_LOGFILTER]\n")
        parser.add_option(
            "--logging-clear-handlers", action="store_true",
            default=False, dest="logcapture_clear",
            help="Clear all other logging handlers")
        parser.add_option(
            "--logging-level", action="store",
            default='NOTSET', dest="logcapture_level",
            help="Set the log level to capture")

    def configure(self, options, conf):
        """Configure plugin.
        """
        self.conf = conf
        # Disable if explicitly disabled, or if logging is
        # configured via logging config file
        if not options.logcapture or conf.loggingConfig:
            self.enabled = False
        self.logformat = options.logcapture_format
        self.logdatefmt = options.logcapture_datefmt
        self.clear = options.logcapture_clear
        self.loglevel = options.logcapture_level
```

```
        if options.logcapture_filters:
            self.filters = options.logcapture_filters.split(',')

    def setupLoghandler(self):
        # setup our handler with root logger
        root_logger = logging.getLogger()
        if self.clear:
            if hasattr(root_logger, "handlers"):
                for handler in root_logger.handlers:
                    root_logger.removeHandler(handler)
            for logger in logging.Logger.manager.loggerDict.values():
                if hasattr(logger, "handlers"):
                    for handler in logger.handlers:
                        logger.removeHandler(handler)
        # make sure there isn't one already
        # you can't simply use "if self.handler not in root_logger.handlers"
        # since at least in unit tests this doesn't work --
        # LogCapture() is instantiated for each test case while root_logger
        # is module global
        # so we always add new MyMemoryHandler instance
        for handler in root_logger.handlers[:]:
            if isinstance(handler, MyMemoryHandler):
                root_logger.handlers.remove(handler)
        root_logger.addHandler(self.handler)
        # Also patch any non-propagating loggers in the tree
        for logger in logging.Logger.manager.loggerDict.values():
            if not getattr(logger, 'propagate', True) and hasattr(logger, "addHandler
↪"):
                for handler in logger.handlers[:]:
                    if isinstance(handler, MyMemoryHandler):
                        logger.handlers.remove(handler)
                logger.addHandler(self.handler)
        # to make sure everything gets captured
        loglevel = getattr(self, "loglevel", "NOTSET")
        root_logger.setLevel(getattr(logging, loglevel))

    def begin(self):
        """Set up logging handler before test run begins.
        """
        self.start()

    def start(self):
        self.handler = MyMemoryHandler(self.logformat, self.logdatefmt,
                                       self.filters)
        self.setupLoghandler()

    def end(self):
        pass

    def beforeTest(self, test):
        """Clear buffers and handlers before test.
        """
        self.setupLoghandler()

    def afterTest(self, test):
        """Clear buffers after test.
        """
        self.handler.truncate()
```

```
    def formatFailure(self, test, err):
        """Add captured log messages to failure output.
        """
        return self.formatError(test, err)

    def formatError(self, test, err):
        """Add captured log messages to error output.
        """
        # logic flow copied from Capture.formatError
        test.capturedLogging = records = self.formatLogRecords()
        if not records:
            return err
        ec, ev, tb = err
        return (ec, self.addCaptureToErr(ev, records), tb)

    def formatLogRecords(self):
        return map(safe_str, self.handler.buffer)

    def addCaptureToErr(self, ev, records):
        return '\n'.join([safe_str(ev), ln('>> begin captured logging <<')] + \
                         records + \
                         [ln('>> end captured logging <<')])
```

Multiprocess: parallel testing

Overview

The multiprocess plugin enables you to distribute your test run among a set of worker processes that run tests in parallel. This can speed up CPU-bound test runs (as long as the number of work processeses is around the number of processors or cores available), but is mainly useful for IO-bound tests that spend most of their time waiting for data to arrive from someplace else.

Note: See *Parallel Testing with nose* for additional documentation and examples. Use of this plugin on python 2.5 or earlier requires the multiprocessing module, also available from PyPI.

How tests are distributed

The ideal case would be to dispatch each test to a worker process separately. This ideal is not attainable in all cases, however, because many test suites depend on context (class, module or package) fixtures.

The plugin can't know (unless you tell it – see below!) if a context fixture can be called many times concurrently (is re-entrant), or if it can be shared among tests running in different processes. Therefore, if a context has fixtures, the default behavior is to dispatch the entire suite to a worker as a unit.

Controlling distribution

There are two context-level variables that you can use to control this default behavior.

If a context's fixtures are re-entrant, set `_multiprocess_can_split_ = True` in the context, and the plugin will dispatch tests in suites bound to that context as if the context had no fixtures. This means that the fixtures will execute concurrently and multiple times, typically once per test.

If a context's fixtures can be shared by tests running in different processes – such as a package-level fixture that starts an external http server or initializes a shared database – then set `_multiprocess_shared_ = True` in the context. These fixtures will then execute in the primary nose process, and tests in those contexts will be individually dispatched to run in parallel.

How results are collected and reported

As each test or suite executes in a worker process, results (failures, errors, and specially handled exceptions like SkipTest) are collected in that process. When the worker process finishes, it returns results to the main nose process. There, any progress output is printed (dots!), and the results from the test run are combined into a consolidated result set. When results have been received for all dispatched tests, or all workers have died, the result summary is output as normal.

Beware!

Not all test suites will benefit from, or even operate correctly using, this plugin. For example, CPU-bound tests will run more slowly if you don't have multiple processors. There are also some differences in plugin interactions and behaviors due to the way in which tests are dispatched and loaded. In general, test loading under this plugin operates as if it were always in directed mode instead of discovered mode. For instance, doctests in test modules will always be found when using this plugin with the doctest plugin.

But the biggest issue you will face is probably concurrency. Unless you have kept your tests as religiously pure unit tests, with no side-effects, no ordering issues, and no external dependencies, chances are you will experience odd, intermittent and unexplainable failures and errors when using this plugin. This doesn't necessarily mean the plugin is broken; it may mean that your test suite is not safe for concurrency.

New Features in 1.1.0

- functions generated by test generators are now added to the worker queue making them multi-threaded.
- fixed timeout functionality, now functions will be terminated with a TimedOutException exception when they exceed their execution time. The worker processes are not terminated.
- added `--process-restartworker` option to restart workers once they are done, this helps control memory usage. Sometimes memory leaks can accumulate making long runs very difficult.
- added global _instantiate_plugins to configure which plugins are started on the worker processes.

Options

`--processes`=NUM
: Spread test run among this many processes. Set a number equal to the number of processors or cores in your machine for best results. Pass a negative number to have the number of processes automatically set to the number of cores. Passing 0 means to disable parallel testing. Default is 0 unless NOSE_PROCESSES is set. [NOSE_PROCESSES]

`--process-timeout`=SECONDS
: Set timeout for return of results from each test runner process. Default is 10. [NOSE_PROCESS_TIMEOUT]

--process-restartworker
If set, will restart each worker process once their tests are done, this helps control memory leaks from killing the system. [NOSE_PROCESS_RESTARTWORKER]

Plugin

class nose.plugins.multiprocess.**MultiProcess**
Bases: *nose.plugins.base.Plugin*

Run tests in multiple processes. Requires processing module.

configure(*options*, *config*)
Configure plugin.

options(*parser*, *env*)
Register command-line options.

prepareTestLoader(*loader*)
Remember loader class so MultiProcessTestRunner can instantiate the right loader.

prepareTestRunner(*runner*)
Replace test runner with MultiProcessTestRunner.

Source

```
"""
Overview
========

The multiprocess plugin enables you to distribute your test run among a set of
worker processes that run tests in parallel. This can speed up CPU-bound test
runs (as long as the number of work processeses is around the number of
processors or cores available), but is mainly useful for IO-bound tests that
spend most of their time waiting for data to arrive from someplace else.

.. note ::

   See :doc:`../doc_tests/test_multiprocess/multiprocess` for
   additional documentation and examples. Use of this plugin on python
   2.5 or earlier requires the multiprocessing_ module, also available
   from PyPI.

.. _multiprocessing : http://code.google.com/p/python-multiprocessing/

How tests are distributed
=========================

The ideal case would be to dispatch each test to a worker process
separately. This ideal is not attainable in all cases, however, because many
test suites depend on context (class, module or package) fixtures.

The plugin can't know (unless you tell it -- see below!) if a context fixture
can be called many times concurrently (is re-entrant), or if it can be shared
among tests running in different processes. Therefore, if a context has
fixtures, the default behavior is to dispatch the entire suite to a worker as
a unit.
```

```
Controlling distribution
^^^^^^^^^^^^^^^^^^^^^^^^

There are two context-level variables that you can use to control this default
behavior.

If a context's fixtures are re-entrant, set ``_multiprocess_can_split_ = True``
in the context, and the plugin will dispatch tests in suites bound to that
context as if the context had no fixtures. This means that the fixtures will
execute concurrently and multiple times, typically once per test.

If a context's fixtures can be shared by tests running in different processes
-- such as a package-level fixture that starts an external http server or
initializes a shared database -- then set ``_multiprocess_shared_ = True`` in
the context. These fixtures will then execute in the primary nose process, and
tests in those contexts will be individually dispatched to run in parallel.

How results are collected and reported
======================================

As each test or suite executes in a worker process, results (failures, errors,
and specially handled exceptions like SkipTest) are collected in that
process. When the worker process finishes, it returns results to the main
nose process. There, any progress output is printed (dots!), and the
results from the test run are combined into a consolidated result
set. When results have been received for all dispatched tests, or all
workers have died, the result summary is output as normal.

Beware!
=======

Not all test suites will benefit from, or even operate correctly using, this
plugin. For example, CPU-bound tests will run more slowly if you don't have
multiple processors. There are also some differences in plugin
interactions and behaviors due to the way in which tests are dispatched and
loaded. In general, test loading under this plugin operates as if it were
always in directed mode instead of discovered mode. For instance, doctests
in test modules will always be found when using this plugin with the doctest
plugin.

But the biggest issue you will face is probably concurrency. Unless you
have kept your tests as religiously pure unit tests, with no side-effects, no
ordering issues, and no external dependencies, chances are you will experience
odd, intermittent and unexplainable failures and errors when using this
plugin. This doesn't necessarily mean the plugin is broken; it may mean that
your test suite is not safe for concurrency.

New Features in 1.1.0
=====================

* functions generated by test generators are now added to the worker queue
  making them multi-threaded.
* fixed timeout functionality, now functions will be terminated with a
  TimedOutException exception when they exceed their execution time. The
  worker processes are not terminated.
* added ``--process-restartworker`` option to restart workers once they are
  done, this helps control memory usage. Sometimes memory leaks can accumulate
  making long runs very difficult.
```

```
* added global _instantiate_plugins to configure which plugins are started
  on the worker processes.

"""

import logging
import os
import sys
import time
import traceback
import unittest
import pickle
import signal
import nose.case
from nose.core import TextTestRunner
from nose import failure
from nose import loader
from nose.plugins.base import Plugin
from nose.pyversion import bytes_
from nose.result import TextTestResult
from nose.suite import ContextSuite
from nose.util import test_address
try:
    # 2.7+
    from unittest.runner import _WritelnDecorator
except ImportError:
    from unittest import _WritelnDecorator
from Queue import Empty
from warnings import warn
try:
    from cStringIO import StringIO
except ImportError:
    import StringIO

# this is a list of plugin classes that will be checked for and created inside
# each worker process
_instantiate_plugins = None

log = logging.getLogger(__name__)

Process = Queue = Pool = Event = Value = Array = None

# have to inherit KeyboardInterrupt to it will interrupt process properly
class TimedOutException(KeyboardInterrupt):
    def __init__(self, value = "Timed Out"):
        self.value = value
    def __str__(self):
        return repr(self.value)

def _import_mp():
    global Process, Queue, Pool, Event, Value, Array
    try:
        from multiprocessing import Manager, Process
        #prevent the server process created in the manager which holds Python
        #objects and allows other processes to manipulate them using proxies
        #to interrupt on SIGINT (keyboardinterrupt) so that the communication
        #channel between subprocesses and main process is still usable after
        #ctrl+C is received in the main process.
```

```
        old=signal.signal(signal.SIGINT, signal.SIG_IGN)
        m = Manager()
        #reset it back so main process will receive a KeyboardInterrupt
        #exception on ctrl+c
        signal.signal(signal.SIGINT, old)
        Queue, Pool, Event, Value, Array = (
                m.Queue, m.Pool, m.Event, m.Value, m.Array
        )
    except ImportError:
        warn("multiprocessing module is not available, multiprocess plugin "
             "cannot be used", RuntimeWarning)


class TestLet:
    def __init__(self, case):
        try:
            self._id = case.id()
        except AttributeError:
            pass
        self._short_description = case.shortDescription()
        self._str = str(case)

    def id(self):
        return self._id

    def shortDescription(self):
        return self._short_description

    def __str__(self):
        return self._str

class MultiProcess(Plugin):
    """
    Run tests in multiple processes. Requires processing module.
    """
    score = 1000
    status = {}

    def options(self, parser, env):
        """
        Register command-line options.
        """
        parser.add_option("--processes", action="store",
                          default=env.get('NOSE_PROCESSES', 0),
                          dest="multiprocess_workers",
                          metavar="NUM",
                          help="Spread test run among this many processes. "
                          "Set a number equal to the number of processors "
                          "or cores in your machine for best results. "
                          "Pass a negative number to have the number of "
                          "processes automatically set to the number of "
                          "cores. Passing 0 means to disable parallel "
                          "testing. Default is 0 unless NOSE_PROCESSES is "
                          "set. "
                          "[NOSE_PROCESSES]")
        parser.add_option("--process-timeout", action="store",
                          default=env.get('NOSE_PROCESS_TIMEOUT', 10),
                          dest="multiprocess_timeout",
```

```
                        metavar="SECONDS",
                        help="Set timeout for return of results from each "
                        "test runner process. Default is 10. "
                        "[NOSE_PROCESS_TIMEOUT]")
        parser.add_option("--process-restartworker", action="store_true",
                        default=env.get('NOSE_PROCESS_RESTARTWORKER', False),
                        dest="multiprocess_restartworker",
                        help="If set, will restart each worker process once"
                        " their tests are done, this helps control memory "
                        "leaks from killing the system. "
                        "[NOSE_PROCESS_RESTARTWORKER]")

    def configure(self, options, config):
        """
        Configure plugin.
        """
        try:
            self.status.pop('active')
        except KeyError:
            pass
        if not hasattr(options, 'multiprocess_workers'):
            self.enabled = False
            return
        # don't start inside of a worker process
        if config.worker:
            return
        self.config = config
        try:
            workers = int(options.multiprocess_workers)
        except (TypeError, ValueError):
            workers = 0
        if workers:
            _import_mp()
            if Process is None:
                self.enabled = False
                return
            # Negative number of workers will cause multiprocessing to hang.
            # Set the number of workers to the CPU count to avoid this.
            if workers < 0:
                try:
                    import multiprocessing
                    workers = multiprocessing.cpu_count()
                except NotImplementedError:
                    self.enabled = False
                    return
            self.enabled = True
            self.config.multiprocess_workers = workers
            t = float(options.multiprocess_timeout)
            self.config.multiprocess_timeout = t
            r = int(options.multiprocess_restartworker)
            self.config.multiprocess_restartworker = r
            self.status['active'] = True

    def prepareTestLoader(self, loader):
        """Remember loader class so MultiProcessTestRunner can instantiate
        the right loader.
        """
        self.loaderClass = loader.__class__
```

```
    def prepareTestRunner(self, runner):
        """Replace test runner with MultiProcessTestRunner.
        """
        # replace with our runner class
        return MultiProcessTestRunner(stream=runner.stream,
                                      verbosity=self.config.verbosity,
                                      config=self.config,
                                      loaderClass=self.loaderClass)

def signalhandler(sig, frame):
    raise TimedOutException()

class MultiProcessTestRunner(TextTestRunner):
    waitkilltime = 5.0 # max time to wait to terminate a process that does not
                       # respond to SIGILL
    def __init__(self, **kw):
        self.loaderClass = kw.pop('loaderClass', loader.defaultTestLoader)
        super(MultiProcessTestRunner, self).__init__(**kw)

    def collect(self, test, testQueue, tasks, to_teardown, result):
        # dispatch and collect results
        # put indexes only on queue because tests aren't picklable
        for case in self.nextBatch(test):
            log.debug("Next batch %s (%s)", case, type(case))
            if (isinstance(case, nose.case.Test) and
                isinstance(case.test, failure.Failure)):
                log.debug("Case is a Failure")
                case(result) # run here to capture the failure
                continue
            # handle shared fixtures
            if isinstance(case, ContextSuite) and case.context is failure.Failure:
                log.debug("Case is a Failure")
                case(result) # run here to capture the failure
                continue
            elif isinstance(case, ContextSuite) and self.sharedFixtures(case):
                log.debug("%s has shared fixtures", case)
                try:
                    case.setUp()
                except (KeyboardInterrupt, SystemExit):
                    raise
                except:
                    log.debug("%s setup failed", sys.exc_info())
                    result.addError(case, sys.exc_info())
                else:
                    to_teardown.append(case)
                    if case.factory:
                        ancestors=case.factory.context.get(case, [])
                        for an in ancestors[:2]:
                            #log.debug('reset ancestor %s', an)
                            if getattr(an, '_multiprocess_shared_', False):
                                an._multiprocess_can_split_=True
                            #an._multiprocess_shared_=False
                    self.collect(case, testQueue, tasks, to_teardown, result)

            else:
                test_addr = self.addtask(testQueue,tasks,case)
                log.debug("Queued test %s (%s) to %s",
```

```
                        len(tasks), test_addr, testQueue)

    def startProcess(self, iworker, testQueue, resultQueue, shouldStop, result):
        currentaddr = Value('c',bytes_(''))
        currentstart = Value('d',time.time())
        keyboardCaught = Event()
        p = Process(target=runner,
                    args=(iworker, testQueue,
                          resultQueue,
                          currentaddr,
                          currentstart,
                          keyboardCaught,
                          shouldStop,
                          self.loaderClass,
                          result.__class__,
                          pickle.dumps(self.config)))
        p.currentaddr = currentaddr
        p.currentstart = currentstart
        p.keyboardCaught = keyboardCaught
        old = signal.signal(signal.SIGILL, signalhandler)
        p.start()
        signal.signal(signal.SIGILL, old)
        return p

    def run(self, test):
        """
        Execute the test (which may be a test suite). If the test is a suite,
        distribute it out among as many processes as have been configured, at
        as fine a level as is possible given the context fixtures defined in
        the suite or any sub-suites.

        """
        log.debug("%s.run(%s) (%s)", self, test, os.getpid())
        wrapper = self.config.plugins.prepareTest(test)
        if wrapper is not None:
            test = wrapper

        # plugins can decorate or capture the output stream
        wrapped = self.config.plugins.setOutputStream(self.stream)
        if wrapped is not None:
            self.stream = wrapped

        testQueue = Queue()
        resultQueue = Queue()
        tasks = []
        completed = []
        workers = []
        to_teardown = []
        shouldStop = Event()

        result = self._makeResult()
        start = time.time()

        self.collect(test, testQueue, tasks, to_teardown, result)

        log.debug("Starting %s workers", self.config.multiprocess_workers)
        for i in range(self.config.multiprocess_workers):
            p = self.startProcess(i, testQueue, resultQueue, shouldStop, result)
```

```
            workers.append(p)
            log.debug("Started worker process %s", i+1)

        total_tasks = len(tasks)
        # need to keep track of the next time to check for timeouts in case
        # more than one process times out at the same time.
        nexttimeout=self.config.multiprocess_timeout
        thrownError = None

        try:
            while tasks:
                log.debug("Waiting for results (%s/%s tasks), next timeout=%.3fs",
                          len(completed), total_tasks,nexttimeout)
                try:
                    iworker, addr, newtask_addrs, batch_result = resultQueue.get(
                                                            timeout=nexttimeout)
                    log.debug('Results received for worker %d, %s, new tasks: %d',
                              iworker,addr,len(newtask_addrs))
                    try:
                        try:
                            tasks.remove(addr)
                        except ValueError:
                            log.warn('worker %s failed to remove from tasks: %s',
                                     iworker,addr)
                        total_tasks += len(newtask_addrs)
                        tasks.extend(newtask_addrs)
                    except KeyError:
                        log.debug("Got result for unknown task? %s", addr)
                        log.debug("current: %s",str(list(tasks)[0]))
                    else:
                        completed.append([addr,batch_result])
                    self.consolidate(result, batch_result)
                    if (self.config.stopOnError
                        and not result.wasSuccessful()):
                        # set the stop condition
                        shouldStop.set()
                        break
                    if self.config.multiprocess_restartworker:
                        log.debug('joining worker %s',iworker)
                        # wait for working, but not that important if worker
                        # cannot be joined in fact, for workers that add to
                        # testQueue, they will not terminate until all their
                        # items are read
                        workers[iworker].join(timeout=1)
                        if not shouldStop.is_set() and not testQueue.empty():
                            log.debug('starting new process on worker %s',iworker)
                            workers[iworker] = self.startProcess(iworker, testQueue,␣
↪resultQueue, shouldStop, result)
                except Empty:
                    log.debug("Timed out with %s tasks pending "
                              "(empty testQueue=%r): %s",
                              len(tasks),testQueue.empty(),str(tasks))
                    any_alive = False
                    for iworker, w in enumerate(workers):
                        if w.is_alive():
                            worker_addr = bytes_(w.currentaddr.value,'ascii')
                            timeprocessing = time.time() - w.currentstart.value
                            if ( len(worker_addr) == 0
```

```
                                    and timeprocessing > self.config.multiprocess_
↪timeout-0.1):
                                log.debug('worker %d has finished its work item, '
                                          'but is not exiting? do we wait for it?',
                                          iworker)
                            else:
                                any_alive = True
                            if (len(worker_addr) > 0
                                and timeprocessing > self.config.multiprocess_timeout-
↪0.1):
                                log.debug('timed out worker %s: %s',
                                          iworker,worker_addr)
                                w.currentaddr.value = bytes_('')
                                # If the process is in C++ code, sending a SIGILL
                                # might not send a python KeybordInterrupt exception
                                # therefore, send multiple signals until an
                                # exception is caught. If this takes too long, then
                                # terminate the process
                                w.keyboardCaught.clear()
                                startkilltime = time.time()
                                while not w.keyboardCaught.is_set() and w.is_alive():
                                    if time.time()-startkilltime > self.waitkilltime:
                                        # have to terminate...
                                        log.error("terminating worker %s",iworker)
                                        w.terminate()
                                        # there is a small probability that the
                                        # terminated process might send a result,
                                        # which has to be specially handled or
                                        # else processes might get orphaned.
                                        workers[iworker] = w = self.
↪startProcess(iworker, testQueue, resultQueue, shouldStop, result)
                                        break
                                    os.kill(w.pid, signal.SIGILL)
                                    time.sleep(0.1)
                    if not any_alive and testQueue.empty():
                        log.debug("All workers dead")
                        break
                nexttimeout=self.config.multiprocess_timeout
                for w in workers:
                    if w.is_alive() and len(w.currentaddr.value) > 0:
                        timeprocessing = time.time()-w.currentstart.value
                        if timeprocessing <= self.config.multiprocess_timeout:
                            nexttimeout = min(nexttimeout,
                                self.config.multiprocess_timeout-timeprocessing)
            log.debug("Completed %s tasks (%s remain)", len(completed), len(tasks))

        except (KeyboardInterrupt, SystemExit), e:
            log.info('parent received ctrl-c when waiting for test results')
            thrownError = e
            #resultQueue.get(False)

            result.addError(test, sys.exc_info())

        try:
            for case in to_teardown:
                log.debug("Tearing down shared fixtures for %s", case)
                try:
                    case.tearDown()
```

```
                except (KeyboardInterrupt, SystemExit):
                    raise
                except:
                    result.addError(case, sys.exc_info())

            stop = time.time()

            # first write since can freeze on shutting down processes
            result.printErrors()
            result.printSummary(start, stop)
            self.config.plugins.finalize(result)

            if thrownError is None:
                log.debug("Tell all workers to stop")
                for w in workers:
                    if w.is_alive():
                        testQueue.put('STOP', block=False)

            # wait for the workers to end
            for iworker,worker in enumerate(workers):
                if worker.is_alive():
                    log.debug('joining worker %s',iworker)
                    worker.join()
                    if worker.is_alive():
                        log.debug('failed to join worker %s',iworker)
        except (KeyboardInterrupt, SystemExit):
            log.info('parent received ctrl-c when shutting down: stop all processes')
            for worker in workers:
                if worker.is_alive():
                    worker.terminate()

            if thrownError: raise thrownError
            else: raise

        return result

    def addtask(testQueue,tasks,case):
        arg = None
        if isinstance(case,nose.case.Test) and hasattr(case.test,'arg'):
            # this removes the top level descriptor and allows real function
            # name to be returned
            case.test.descriptor = None
            arg = case.test.arg
        test_addr = MultiProcessTestRunner.address(case)
        testQueue.put((test_addr,arg), block=False)
        if arg is not None:
            test_addr += str(arg)
        if tasks is not None:
            tasks.append(test_addr)
        return test_addr
    addtask = staticmethod(addtask)

    def address(case):
        if hasattr(case, 'address'):
            file, mod, call = case.address()
        elif hasattr(case, 'context'):
            file, mod, call = test_address(case.context)
        else:
```

```
            raise Exception("Unable to convert %s to address" % case)
        parts = []
        if file is None:
            if mod is None:
                raise Exception("Unaddressable case %s" % case)
            else:
                parts.append(mod)
        else:
            # strip __init__.py(c) from end of file part
            # if present, having it there confuses loader
            dirname, basename = os.path.split(file)
            if basename.startswith('__init__'):
                file = dirname
            parts.append(file)
        if call is not None:
            parts.append(call)
        return ':'.join(map(str, parts))
    address = staticmethod(address)

    def nextBatch(self, test):
        # allows tests or suites to mark themselves as not safe
        # for multiprocess execution
        if hasattr(test, 'context'):
            if not getattr(test.context, '_multiprocess_', True):
                return

        if ((isinstance(test, ContextSuite)
             and test.hasFixtures(self.checkCanSplit))
            or not getattr(test, 'can_split', True)
            or not isinstance(test, unittest.TestSuite)):
            # regular test case, or a suite with context fixtures

            # special case: when run like nosetests path/to/module.py
            # the top-level suite has only one item, and it shares
            # the same context as that item. In that case, we want the
            # item, not the top-level suite
            if isinstance(test, ContextSuite):
                contained = list(test)
                if (len(contained) == 1
                    and getattr(contained[0],
                                'context', None) == test.context):
                    test = contained[0]
            yield test
        else:
            # Suite is without fixtures at this level; but it may have
            # fixtures at any deeper level, so we need to examine it all
            # the way down to the case level
            for case in test:
                for batch in self.nextBatch(case):
                    yield batch

    def checkCanSplit(context, fixt):
        """
        Callback that we use to check whether the fixtures found in a
        context or ancestor are ones we care about.

        Contexts can tell us that their fixtures are reentrant by setting
        _multiprocess_can_split_. So if we see that, we return False to
```

```
        disregard those fixtures.
        """
        if not fixt:
            return False
        if getattr(context, '_multiprocess_can_split_', False):
            return False
        return True
    checkCanSplit = staticmethod(checkCanSplit)

    def sharedFixtures(self, case):
        context = getattr(case, 'context', None)
        if not context:
            return False
        return getattr(context, '_multiprocess_shared_', False)

    def consolidate(self, result, batch_result):
        log.debug("batch result is %s" , batch_result)
        try:
            output, testsRun, failures, errors, errorClasses = batch_result
        except ValueError:
            log.debug("result in unexpected format %s", batch_result)
            failure.Failure(*sys.exc_info())(result)
            return
        self.stream.write(output)
        result.testsRun += testsRun
        result.failures.extend(failures)
        result.errors.extend(errors)
        for key, (storage, label, isfail) in errorClasses.items():
            if key not in result.errorClasses:
                # Ordinarily storage is result attribute
                # but it's only processed through the errorClasses
                # dict, so it's ok to fake it here
                result.errorClasses[key] = ([], label, isfail)
            mystorage, _junk, _junk = result.errorClasses[key]
            mystorage.extend(storage)
        log.debug("Ran %s tests (total: %s)", testsRun, result.testsRun)


def runner(ix, testQueue, resultQueue, currentaddr, currentstart,
           keyboardCaught, shouldStop, loaderClass, resultClass, config):
    try:
        try:
            return __runner(ix, testQueue, resultQueue, currentaddr, currentstart,
                    keyboardCaught, shouldStop, loaderClass, resultClass, config)
        except KeyboardInterrupt:
            log.debug('Worker %s keyboard interrupt, stopping',ix)
    except Empty:
        log.debug("Worker %s timed out waiting for tasks", ix)

def __runner(ix, testQueue, resultQueue, currentaddr, currentstart,
           keyboardCaught, shouldStop, loaderClass, resultClass, config):

    config = pickle.loads(config)
    dummy_parser = config.parserClass()
    if _instantiate_plugins is not None:
        for pluginclass in _instantiate_plugins:
            plugin = pluginclass()
            plugin.addOptions(dummy_parser,{})
```

```
            config.plugins.addPlugin(plugin)
    config.plugins.configure(config.options,config)
    config.plugins.begin()
    log.debug("Worker %s executing, pid=%d", ix,os.getpid())

    def get():
        return testQueue.get(timeout=config.multiprocess_timeout)

    def makeResult():
        stream = _WritelnDecorator(StringIO())
        result = resultClass(stream, descriptions=1,
                             verbosity=config.verbosity,
                             config=config)
        plug_result = config.plugins.prepareTestResult(result)
        if plug_result:
            return plug_result
        return result

    def batch(result):
        failures = [(TestLet(c), err) for c, err in result.failures]
        errors = [(TestLet(c), err) for c, err in result.errors]
        errorClasses = {}
        for key, (storage, label, isfail) in result.errorClasses.items():
            errorClasses[key] = ([(TestLet(c), err) for c, err in storage],
                                 label, isfail)
        return (
            result.stream.getvalue(),
            result.testsRun,
            failures,
            errors,
            errorClasses)
    for test_addr, arg in iter(get, 'STOP'):
        if shouldStop.is_set():
            log.exception('Worker %d STOPPED',ix)
            break
        result = makeResult()
        loader = loaderClass(config=config)
        loader.suiteClass.suiteClass = NoSharedFixtureContextSuite
        test = loader.loadTestsFromNames([test_addr])
        test.testQueue = testQueue
        test.tasks = []
        test.arg = arg
        log.debug("Worker %s Test is %s (%s)", ix, test_addr, test)
        try:
            if arg is not None:
                test_addr = test_addr + str(arg)
            currentaddr.value = bytes_(test_addr)
            currentstart.value = time.time()
            test(result)
            currentaddr.value = bytes_('')
            resultQueue.put((ix, test_addr, test.tasks, batch(result)))
        except KeyboardInterrupt, e: #TimedOutException:
            timeout = isinstance(e, TimedOutException)
            if timeout:
                keyboardCaught.set()
            if len(currentaddr.value):
                if timeout:
                    msg = 'Worker %s timed out, failing current test %s'
```

```
                else:
                    msg = 'Worker %s keyboard interrupt, failing current test %s'
                log.exception(msg,ix,test_addr)
                currentaddr.value = bytes_('')
                failure.Failure(*sys.exc_info())(result)
                resultQueue.put((ix, test_addr, test.tasks, batch(result)))
            else:
                if timeout:
                    msg = 'Worker %s test %s timed out'
                else:
                    msg = 'Worker %s test %s keyboard interrupt'
                log.debug(msg,ix,test_addr)
                resultQueue.put((ix, test_addr, test.tasks, batch(result)))
            if not timeout:
                raise
        except SystemExit:
            currentaddr.value = bytes_('')
            log.exception('Worker %s system exit',ix)
            raise
        except:
            currentaddr.value = bytes_('')
            log.exception("Worker %s error running test or returning "
                          "results",ix)
            failure.Failure(*sys.exc_info())(result)
            resultQueue.put((ix, test_addr, test.tasks, batch(result)))
        if config.multiprocess_restartworker:
            break
    log.debug("Worker %s ending", ix)


class NoSharedFixtureContextSuite(ContextSuite):
    """
    Context suite that never fires shared fixtures.

    When a context sets _multiprocess_shared_, fixtures in that context
    are executed by the main process. Using this suite class prevents them
    from executing in the runner process as well.

    """
    testQueue = None
    tasks = None
    arg = None
    def setupContext(self, context):
        if getattr(context, '_multiprocess_shared_', False):
            return
        super(NoSharedFixtureContextSuite, self).setupContext(context)

    def teardownContext(self, context):
        if getattr(context, '_multiprocess_shared_', False):
            return
        super(NoSharedFixtureContextSuite, self).teardownContext(context)
    def run(self, result):
        """Run tests in suite inside of suite fixtures.
        """
        # proxy the result for myself
        log.debug("suite %s (%s) run called, tests: %s",
                  id(self), self, self._tests)
        if self.resultProxy:
```

```
            result, orig = self.resultProxy(result, self), result
        else:
            result, orig = result, result
        try:
            #log.debug('setUp for %s', id(self));
            self.setUp()
        except KeyboardInterrupt:
            raise
        except:
            self.error_context = 'setup'
            result.addError(self, self._exc_info())
            return
        try:
            for test in self._tests:
                if (isinstance(test,nose.case.Test)
                    and self.arg is not None):
                    test.test.arg = self.arg
                else:
                    test.arg = self.arg
                test.testQueue = self.testQueue
                test.tasks = self.tasks
                if result.shouldStop:
                    log.debug("stopping")
                    break
                # each nose.case.Test will create its own result proxy
                # so the cases need the original result, to avoid proxy
                # chains
                #log.debug('running test %s in suite %s', test, self);
                try:
                    test(orig)
                except KeyboardInterrupt, e:
                    timeout = isinstance(e, TimedOutException)
                    if timeout:
                        msg = 'Timeout when running test %s in suite %s'
                    else:
                        msg = 'KeyboardInterrupt when running test %s in suite %s'
                    log.debug(msg, test, self)
                    err = (TimedOutException,TimedOutException(str(test)),
                           sys.exc_info()[2])
                    test.config.plugins.addError(test,err)
                    orig.addError(test,err)
                    if not timeout:
                        raise
        finally:
            self.has_run = True
            try:
                #log.debug('tearDown for %s', id(self));
                self.tearDown()
            except KeyboardInterrupt:
                raise
            except:
                self.error_context = 'teardown'
                result.addError(self, self._exc_info())
```

Prof: enable profiling using the hotshot profiler

This plugin will run tests using the hotshot profiler, which is part of the standard library. To turn it on, use the `--with-profile` option or set the NOSE_WITH_PROFILE environment variable. Profiler output can be controlled with the `--profile-sort` and `--profile-restrict` options, and the profiler output file may be changed with `--profile-stats-file`.

See the hotshot documentation in the standard library documentation for more details on the various output options.

Options

`--with-profile`
Enable plugin Profile: Use this plugin to run tests using the hotshot profiler. [NOSE_WITH_PROFILE]

`--profile-sort`=SORT
Set sort order for profiler output

`--profile-stats-file`=FILE
Profiler stats file; default is a new temp file on each run

`--profile-restrict`=RESTRICT
Restrict profiler output. See help for pstats.Stats for details

Plugin

class `nose.plugins.prof.`**`Profile`**
Bases: *nose.plugins.base.Plugin*

Use this plugin to run tests using the hotshot profiler.

`begin`()
Create profile stats file and load profiler.

`configure`(*options*, *conf*)
Configure plugin.

`finalize`(*result*)
Clean up stats file, if configured to do so.

`options`(*parser*, *env*)
Register commandline options.

`prepareTest`(*test*)
Wrap entire test run in `prof.runcall()`.

`report`(*stream*)
Output profiler report.

Source

```
"""This plugin will run tests using the hotshot profiler, which is part
of the standard library. To turn it on, use the ``--with-profile`` option
or set the NOSE_WITH_PROFILE environment variable. Profiler output can be
controlled with the ``--profile-sort`` and ``--profile-restrict`` options,
and the profiler output file may be changed with ``--profile-stats-file``.
```

```
See the `hotshot documentation`_ in the standard library documentation for
more details on the various output options.

.. _hotshot documentation: http://docs.python.org/library/hotshot.html
"""

try:
    import hotshot
    from hotshot import stats
except ImportError:
    hotshot, stats = None, None
import logging
import os
import sys
import tempfile
from nose.plugins.base import Plugin
from nose.util import tolist

log = logging.getLogger('nose.plugins')

class Profile(Plugin):
    """
    Use this plugin to run tests using the hotshot profiler.
    """
    pfile = None
    clean_stats_file = False
    def options(self, parser, env):
        """Register commandline options.
        """
        if not self.available():
            return
        Plugin.options(self, parser, env)
        parser.add_option('--profile-sort', action='store', dest='profile_sort',
                          default=env.get('NOSE_PROFILE_SORT', 'cumulative'),
                          metavar="SORT",
                          help="Set sort order for profiler output")
        parser.add_option('--profile-stats-file', action='store',
                          dest='profile_stats_file',
                          metavar="FILE",
                          default=env.get('NOSE_PROFILE_STATS_FILE'),
                          help='Profiler stats file; default is a new '
                          'temp file on each run')
        parser.add_option('--profile-restrict', action='append',
                          dest='profile_restrict',
                          metavar="RESTRICT",
                          default=env.get('NOSE_PROFILE_RESTRICT'),
                          help="Restrict profiler output. See help for "
                          "pstats.Stats for details")

    def available(cls):
        return hotshot is not None
    available = classmethod(available)

    def begin(self):
        """Create profile stats file and load profiler.
        """
        if not self.available():
            return
```

```
        self._create_pfile()
        self.prof = hotshot.Profile(self.pfile)

    def configure(self, options, conf):
        """Configure plugin.
        """
        if not self.available():
            self.enabled = False
            return
        Plugin.configure(self, options, conf)
        self.conf = conf
        if options.profile_stats_file:
            self.pfile = options.profile_stats_file
            self.clean_stats_file = False
        else:
            self.pfile = None
            self.clean_stats_file = True
        self.fileno = None
        self.sort = options.profile_sort
        self.restrict = tolist(options.profile_restrict)

    def prepareTest(self, test):
        """Wrap entire test run in :func:`prof.runcall`.
        """
        if not self.available():
            return
        log.debug('preparing test %s' % test)
        def run_and_profile(result, prof=self.prof, test=test):
            self._create_pfile()
            prof.runcall(test, result)
        return run_and_profile

    def report(self, stream):
        """Output profiler report.
        """
        log.debug('printing profiler report')
        self.prof.close()
        prof_stats = stats.load(self.pfile)
        prof_stats.sort_stats(self.sort)

        # 2.5 has completely different stream handling from 2.4 and earlier.
        # Before 2.5, stats objects have no stream attribute; in 2.5 and later
        # a reference sys.stdout is stored before we can tweak it.
        compat_25 = hasattr(prof_stats, 'stream')
        if compat_25:
            tmp = prof_stats.stream
            prof_stats.stream = stream
        else:
            tmp = sys.stdout
            sys.stdout = stream
        try:
            if self.restrict:
                log.debug('setting profiler restriction to %s', self.restrict)
                prof_stats.print_stats(*self.restrict)
            else:
                prof_stats.print_stats()
        finally:
            if compat_25:
```

```
                prof_stats.stream = tmp
            else:
                sys.stdout = tmp

    def finalize(self, result):
        """Clean up stats file, if configured to do so.
        """
        if not self.available():
            return
        try:
            self.prof.close()
        except AttributeError:
            # TODO: is this trying to catch just the case where not
            # hasattr(self.prof, "close")?  If so, the function call should be
            # moved out of the try: suite.
            pass
        if self.clean_stats_file:
            if self.fileno:
                try:
                    os.close(self.fileno)
                except OSError:
                    pass
            try:
                os.unlink(self.pfile)
            except OSError:
                pass
        return None

    def _create_pfile(self):
        if not self.pfile:
            self.fileno, self.pfile = tempfile.mkstemp()
            self.clean_stats_file = True
```

Skip: mark tests as skipped

This plugin installs a SKIP error class for the SkipTest exception. When SkipTest is raised, the exception will be logged in the skipped attribute of the result, 'S' or 'SKIP' (verbose) will be output, and the exception will not be counted as an error or failure. This plugin is enabled by default but may be disabled with the `--no-skip` option.

Options

--no-skip
Disable special handling of SkipTest exceptions.

Plugin

class nose.plugins.skip.**Skip**
Bases: *nose.plugins.errorclass.ErrorClassPlugin*

Plugin that installs a SKIP error class for the SkipTest exception. When SkipTest is raised, the exception will be logged in the skipped attribute of the result, 'S' or 'SKIP' (verbose) will be output, and the exception will not be counted as an error or failure.

configure (*options*, *conf*)
Configure plugin. Skip plugin is enabled by default.

options (*parser*, *env*)
Add my options to command line.

Source

```
"""
This plugin installs a SKIP error class for the SkipTest exception.
When SkipTest is raised, the exception will be logged in the skipped
attribute of the result, 'S' or 'SKIP' (verbose) will be output, and
the exception will not be counted as an error or failure. This plugin
is enabled by default but may be disabled with the ``--no-skip`` option.
"""

from nose.plugins.errorclass import ErrorClass, ErrorClassPlugin


# on SkipTest:
#  - unittest SkipTest is first preference, but it's only available
#    for >= 2.7
#  - unittest2 SkipTest is second preference for older pythons.  This
#    mirrors logic for choosing SkipTest exception in testtools
#  - if none of the above, provide custom class
try:
    from unittest.case import SkipTest
except ImportError:
    try:
        from unittest2.case import SkipTest
    except ImportError:
        class SkipTest(Exception):
            """Raise this exception to mark a test as skipped.
            """
            pass


class Skip(ErrorClassPlugin):
    """
    Plugin that installs a SKIP error class for the SkipTest
    exception.  When SkipTest is raised, the exception will be logged
    in the skipped attribute of the result, 'S' or 'SKIP' (verbose)
    will be output, and the exception will not be counted as an error
    or failure.
    """
    enabled = True
    skipped = ErrorClass(SkipTest,
                         label='SKIP',
                         isfailure=False)

    def options(self, parser, env):
        """
        Add my options to command line.
        """
        env_opt = 'NOSE_WITHOUT_SKIP'
        parser.add_option('--no-skip', action='store_true',
                          dest='noSkip', default=env.get(env_opt, False),
```

```
                          help="Disable special handling of SkipTest "
                          "exceptions.")

    def configure(self, options, conf):
        """
        Configure plugin. Skip plugin is enabled by default.
        """
        if not self.can_configure:
            return
        self.conf = conf
        disable = getattr(options, 'noSkip', False)
        if disable:
            self.enabled = False
```

Testid: add a test id to each test name output

This plugin adds a test id (like #1) to each test name output. After you've run once to generate test ids, you can re-run individual tests by activating the plugin and passing the ids (with or without the # prefix) instead of test names.

For example, if your normal test run looks like:

```
% nosetests -v
tests.test_a ... ok
tests.test_b ... ok
tests.test_c ... ok
```

When adding `--with-id` you'll see:

```
% nosetests -v --with-id
#1 tests.test_a ... ok
#2 tests.test_b ... ok
#3 tests.test_c ... ok
```

Then you can re-run individual tests by supplying just an id number:

```
% nosetests -v --with-id 2
#2 tests.test_b ... ok
```

You can also pass multiple id numbers:

```
% nosetests -v --with-id 2 3
#2 tests.test_b ... ok
#3 tests.test_c ... ok
```

Since most shells consider '#' a special character, you can leave it out when specifying a test id.

Note that when run without the -v switch, no special output is displayed, but the ids file is still written.

Looping over failed tests

This plugin also adds a mode that will direct the test runner to record failed tests. Subsequent test runs will then run only the tests that failed last time. Activate this mode with the `--failed` switch:

```
% nosetests -v --failed
#1 test.test_a ... ok
#2 test.test_b ... ERROR
#3 test.test_c ... FAILED
#4 test.test_d ... ok
```

On the second run, only tests #2 and #3 will run:

```
% nosetests -v --failed
#2 test.test_b ... ERROR
#3 test.test_c ... FAILED
```

As you correct errors and tests pass, they'll drop out of subsequent runs.

First:

```
% nosetests -v --failed
#2 test.test_b ... ok
#3 test.test_c ... FAILED
```

Second:

```
% nosetests -v --failed
#3 test.test_c ... FAILED
```

When all tests pass, the full set will run on the next invocation.

First:

```
% nosetests -v --failed
#3 test.test_c ... ok
```

Second:

```
% nosetests -v --failed
#1 test.test_a ... ok
#2 test.test_b ... ok
#3 test.test_c ... ok
#4 test.test_d ... ok
```

Note: If you expect to use `--failed` regularly, it's a good idea to always run using the `--with-id` option. This will ensure that an id file is always created, allowing you to add `--failed` to the command line as soon as you have failing tests. Otherwise, your first run using `--failed` will (perhaps surprisingly) run *all* tests, because there won't be an id file containing the record of failed tests from your previous run.

Options

`--with-id`
: Enable plugin TestId: Activate to add a test id (like #1) to each test name output. Activate with –failed to rerun failing tests only. [NOSE_WITH_ID]

`--id-file``=FILE`
: Store test ids found in test runs in this file. Default is the file .noseids in the working directory.

--failed
Run the tests that failed in the last test run.

Plugin

class nose.plugins.testid.**TestId**
Bases: *nose.plugins.base.Plugin*

Activate to add a test id (like #1) to each test name output. Activate with –failed to rerun failing tests only.

configure(*options*, *conf*)
Configure plugin.

finalize(*result*)
Save new ids file, if needed.

loadTestsFromNames(*names*, *module=None*)
Translate ids in the list of requested names into their test addresses, if they are found in my dict of tests.

options(*parser*, *env*)
Register commandline options.

setOutputStream(*stream*)
Get handle on output stream so the plugin can print id #s

startTest(*test*)
Maybe output an id # before the test name.

Example output:

```
#1 test.test ... ok
#2 test.test_two ... ok
```

Source

```
"""
This plugin adds a test id (like #1) to each test name output. After
you've run once to generate test ids, you can re-run individual
tests by activating the plugin and passing the ids (with or
without the # prefix) instead of test names.

For example, if your normal test run looks like::

  % nosetests -v
  tests.test_a ... ok
  tests.test_b ... ok
  tests.test_c ... ok

When adding ``--with-id`` you'll see::

  % nosetests -v --with-id
  #1 tests.test_a ... ok
  #2 tests.test_b ... ok
  #3 tests.test_c ... ok

Then you can re-run individual tests by supplying just an id number::
```

```
  % nosetests -v --with-id 2
  #2 tests.test_b ... ok

You can also pass multiple id numbers::

  % nosetests -v --with-id 2 3
  #2 tests.test_b ... ok
  #3 tests.test_c ... ok

Since most shells consider '#' a special character, you can leave it out when
specifying a test id.

Note that when run without the -v switch, no special output is displayed, but
the ids file is still written.

Looping over failed tests
-------------------------

This plugin also adds a mode that will direct the test runner to record
failed tests. Subsequent test runs will then run only the tests that failed
last time. Activate this mode with the ``--failed`` switch::

 % nosetests -v --failed
 #1 test.test_a ... ok
 #2 test.test_b ... ERROR
 #3 test.test_c ... FAILED
 #4 test.test_d ... ok

On the second run, only tests #2 and #3 will run::

 % nosetests -v --failed
 #2 test.test_b ... ERROR
 #3 test.test_c ... FAILED

As you correct errors and tests pass, they'll drop out of subsequent runs.

First::

 % nosetests -v --failed
 #2 test.test_b ... ok
 #3 test.test_c ... FAILED

Second::

 % nosetests -v --failed
 #3 test.test_c ... FAILED

When all tests pass, the full set will run on the next invocation.

First::

 % nosetests -v --failed
 #3 test.test_c ... ok

Second::

 % nosetests -v --failed
 #1 test.test_a ... ok
```

```
 #2 test.test_b ... ok
 #3 test.test_c ... ok
 #4 test.test_d ... ok

.. note ::

  If you expect to use ``--failed`` regularly, it's a good idea to always run
  using the ``--with-id`` option. This will ensure that an id file is always
  created, allowing you to add ``--failed`` to the command line as soon as
  you have failing tests. Otherwise, your first run using ``--failed`` will
  (perhaps surprisingly) run *all* tests, because there won't be an id file
  containing the record of failed tests from your previous run.

"""
__test__ = False

import logging
import os
from nose.plugins import Plugin
from nose.util import src, set

try:
    from cPickle import dump, load
except ImportError:
    from pickle import dump, load

log = logging.getLogger(__name__)


class TestId(Plugin):
    """
    Activate to add a test id (like #1) to each test name output. Activate
    with --failed to rerun failing tests only.
    """
    name = 'id'
    idfile = None
    collecting = True
    loopOnFailed = False

    def options(self, parser, env):
        """Register commandline options.
        """
        Plugin.options(self, parser, env)
        parser.add_option('--id-file', action='store', dest='testIdFile',
                          default='.noseids', metavar="FILE",
                          help="Store test ids found in test runs in this "
                          "file. Default is the file .noseids in the "
                          "working directory.")
        parser.add_option('--failed', action='store_true',
                          dest='failed', default=False,
                          help="Run the tests that failed in the last "
                          "test run.")

    def configure(self, options, conf):
        """Configure plugin.
        """
        Plugin.configure(self, options, conf)
        if options.failed:
```

```
            self.enabled = True
            self.loopOnFailed = True
            log.debug("Looping on failed tests")
        self.idfile = os.path.expanduser(options.testIdFile)
        if not os.path.isabs(self.idfile):
            self.idfile = os.path.join(conf.workingDir, self.idfile)
        self.id = 1
        # Ids and tests are mirror images: ids are {id: test address} and
        # tests are {test address: id}
        self.ids = {}
        self.tests = {}
        self.failed = []
        self.source_names = []
        # used to track ids seen when tests is filled from
        # loaded ids file
        self._seen = {}
        self._write_hashes = conf.verbosity >= 2

    def finalize(self, result):
        """Save new ids file, if needed.
        """
        if result.wasSuccessful():
            self.failed = []
        if self.collecting:
            ids = dict(list(zip(list(self.tests.values()), list(self.tests.keys()))))
        else:
            ids = self.ids
        fh = open(self.idfile, 'wb')
        dump({'ids': ids,
              'failed': self.failed,
              'source_names': self.source_names}, fh)
        fh.close()
        log.debug('Saved test ids: %s, failed %s to %s',
                  ids, self.failed, self.idfile)

    def loadTestsFromNames(self, names, module=None):
        """Translate ids in the list of requested names into their
        test addresses, if they are found in my dict of tests.
        """
        log.debug('ltfn %s %s', names, module)
        try:
            fh = open(self.idfile, 'rb')
            try:
                data = load(fh)
                if 'ids' in data:
                    self.ids = data['ids']
                    self.failed = data['failed']
                    self.source_names = data['source_names']
                else:
                    # old ids field
                    self.ids = data
                    self.failed = []
                    self.source_names = names
                if self.ids:
                    self.id = max(self.ids) + 1
                    self.tests = dict(list(zip(list(self.ids.values()), list(self.ids.
↪keys()))))
                else:
```

```
                    self.id = 1
                log.debug(
                    'Loaded test ids %s tests %s failed %s sources %s from %s',
                    self.ids, self.tests, self.failed, self.source_names,
                    self.idfile)
            except ValueError, e:
                # load() may throw a ValueError when reading the ids file, if it
                # was generated with a newer version of Python than we are currently
                # running.
                log.debug('Error loading %s : %s', self.idfile, str(e))
            finally:
                fh.close()
        except IOError:
            log.debug('IO error reading %s', self.idfile)

        if self.loopOnFailed and self.failed:
            self.collecting = False
            names = self.failed
            self.failed = []
        # I don't load any tests myself, only translate names like '#2'
        # into the associated test addresses
        translated = []
        new_source = []
        really_new = []
        for name in names:
            trans = self.tr(name)
            if trans != name:
                translated.append(trans)
            else:
                new_source.append(name)
        # names that are not ids and that are not in the current
        # list of source names go into the list for next time
        if new_source:
            new_set = set(new_source)
            old_set = set(self.source_names)
            log.debug("old: %s new: %s", old_set, new_set)
            really_new = [s for s in new_source
                          if not s in old_set]
            if really_new:
                # remember new sources
                self.source_names.extend(really_new)
            if not translated:
                # new set of source names, no translations
                # means "run the requested tests"
                names = new_source
        else:
            # no new names to translate and add to id set
            self.collecting = False
        log.debug("translated: %s new sources %s names %s",
                  translated, really_new, names)
        return (None, translated + really_new or names)

    def makeName(self, addr):
        log.debug("Make name %s", addr)
        filename, module, call = addr
        if filename is not None:
            head = src(filename)
        else:
```

```
            head = module
        if call is not None:
            return "%s:%s" % (head, call)
        return head

    def setOutputStream(self, stream):
        """Get handle on output stream so the plugin can print id #s
        """
        self.stream = stream

    def startTest(self, test):
        """Maybe output an id # before the test name.

        Example output::

          #1 test.test ... ok
          #2 test.test_two ... ok

        """
        adr = test.address()
        log.debug('start test %s (%s)', adr, adr in self.tests)
        if adr in self.tests:
            if adr in self._seen:
                self.write('    ')
            else:
                self.write('#%s ' % self.tests[adr])
                self._seen[adr] = 1
            return
        self.tests[adr] = self.id
        self.write('#%s ' % self.id)
        self.id += 1

    def afterTest(self, test):
        # None means test never ran, False means failed/err
        if test.passed is False:
            try:
                key = str(self.tests[test.address()])
            except KeyError:
                # never saw this test -- startTest didn't run
                pass
            else:
                if key not in self.failed:
                    self.failed.append(key)

    def tr(self, name):
        log.debug("tr '%s'", name)
        try:
            key = int(name.replace('#', ''))
        except ValueError:
            return name
        log.debug("Got key %s", key)
        # I'm running tests mapped from the ids file,
        # not collecting new ones
        if key in self.ids:
            return self.makeName(self.ids[key])
        return name

    def write(self, output):
```

```
        if self._write_hashes:
            self.stream.write(output)
```

Xunit: output test results in xunit format

This plugin provides test results in the standard XUnit XML format.

It's designed for the Jenkins (previously Hudson) continuous build system, but will probably work for anything else that understands an XUnit-formatted XML representation of test results.

Add this shell command to your builder

```
nosetests --with-xunit
```

And by default a file named nosetests.xml will be written to the working directory.

In a Jenkins builder, tick the box named "Publish JUnit test result report" under the Post-build Actions and enter this value for Test report XMLs:

```
**/nosetests.xml
```

If you need to change the name or location of the file, you can set the `--xunit-file` option.

If you need to change the name of the test suite, you can set the `--xunit-testsuite-name` option.

If you need to prefix the name of the tested classes, you can set the `--xunit-prefix-with-testsuite-name` option. This can be used in a matrixed build to distinguish between failures in different environments. The testsuite name is used as a prefix.

Here is an abbreviated version of what an XML test report might look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="nosetests" tests="1" errors="1" failures="0" skip="0">
    <testcase classname="path_to_test_suite.TestSomething"
              name="test_it" time="0">
        <error type="exceptions.TypeError" message="oops, wrong type">
        Traceback (most recent call last):
        ...
        TypeError: oops, wrong type
        </error>
    </testcase>
</testsuite>
```

Options

--with-xunit
: Enable plugin Xunit: This plugin provides test results in the standard XUnit XML format. [NOSE_WITH_XUNIT]

--xunit-file=FILE
: Path to xml file to store the xunit report in. Default is nosetests.xml in the working directory [NOSE_XUNIT_FILE]

--xunit-testsuite-name=PACKAGE
: Name of the testsuite in the xunit xml, generated by plugin. Default test suite name is nosetests.

--xunit-prefix-with-testsuite-name
Whether to prefix the class name under test with testsuite name. Defaults to false.

Plugin

class nose.plugins.xunit.**Xunit**
Bases: *nose.plugins.base.Plugin*

This plugin provides test results in the standard XUnit XML format.

addError(*test*, *err*, *capt=None*)
Add error output to Xunit report.

addFailure(*test*, *err*, *capt=None*, *tb_info=None*)
Add failure output to Xunit report.

addSuccess(*test*, *capt=None*)
Add success output to Xunit report.

beforeTest(*test*)
Initializes a timer before starting a test.

configure(*options*, *config*)
Configures the xunit plugin.

options(*parser*, *env*)
Sets additional command line options.

report(*stream*)
Writes an Xunit-formatted XML file

The file includes a report of test errors and failures.

Source

```
"""This plugin provides test results in the standard XUnit XML format.

It's designed for the `Jenkins`_ (previously Hudson) continuous build
system, but will probably work for anything else that understands an
XUnit-formatted XML representation of test results.

Add this shell command to your builder ::

    nosetests --with-xunit

And by default a file named nosetests.xml will be written to the
working directory.

In a Jenkins builder, tick the box named "Publish JUnit test result report"
under the Post-build Actions and enter this value for Test report XMLs::

    **/nosetests.xml

If you need to change the name or location of the file, you can set the
``--xunit-file`` option.

If you need to change the name of the test suite, you can set the
``--xunit-testsuite-name`` option.
```

```
If you need to prefix the name of the tested classes, you can set the
``--xunit-prefix-with-testsuite-name`` option.
This can be used in a matrixed build to distinguish between failures
in different environments.
The testsuite name is used as a prefix.

Here is an abbreviated version of what an XML test report might look like::

    <?xml version="1.0" encoding="UTF-8"?>
    <testsuite name="nosetests" tests="1" errors="1" failures="0" skip="0">
        <testcase classname="path_to_test_suite.TestSomething"
                  name="test_it" time="0">
            <error type="exceptions.TypeError" message="oops, wrong type">
            Traceback (most recent call last):
            ...
            TypeError: oops, wrong type
            </error>
        </testcase>
    </testsuite>

.. _Jenkins: http://jenkins-ci.org/

"""
import codecs
import doctest
import os
import sys
import traceback
import re
import inspect
from StringIO import StringIO
from time import time
from xml.sax import saxutils

from nose.plugins.base import Plugin
from nose.exc import SkipTest
from nose.pyversion import force_unicode, format_exception

# Invalid XML characters, control characters 0-31 sans \t, \n and \r
CONTROL_CHARACTERS = re.compile(r"[\000-\010\013\014\016-\037]")

TEST_ID = re.compile(r'^(.*?)(\(.*\))$')

def xml_safe(value):
    """Replaces invalid XML characters with '?'."""
    return CONTROL_CHARACTERS.sub('?', value)

def escape_cdata(cdata):
    """Escape a string for an XML CDATA section."""
    return xml_safe(cdata).replace(']]>', ']]>]]&gt;<![CDATA[')

def id_split(idval):
    m = TEST_ID.match(idval)
    if m:
        name, fargs = m.groups()
        head, tail = name.rsplit(".", 1)
        return [head, tail+fargs]
```

```
    else:
        return idval.rsplit(".", 1)

def nice_classname(obj):
    """Returns a nice name for class object or class instance.

        >>> nice_classname(Exception()) # doctest: +ELLIPSIS
        '...Exception'
        >>> nice_classname(Exception) # doctest: +ELLIPSIS
        '...Exception'

    """
    if inspect.isclass(obj):
        cls_name = obj.__name__
    else:
        cls_name = obj.__class__.__name__
    mod = inspect.getmodule(obj)
    if mod:
        name = mod.__name__
        # jython
        if name.startswith('org.python.core.'):
            name = name[len('org.python.core.'):]
        return "%s.%s" % (name, cls_name)
    else:
        return cls_name

def exc_message(exc_info):
    """Return the exception's message."""
    exc = exc_info[1]
    if exc is None:
        # str exception
        result = exc_info[0]
    else:
        try:
            result = str(exc)
        except UnicodeEncodeError:
            try:
                result = unicode(exc)
            except UnicodeError:
                # Fallback to args as neither str nor
                # unicode(Exception(u'\xe6')) work in Python < 2.6
                result = exc.args[0]
    result = force_unicode(result, 'UTF-8')
    return xml_safe(result)

class Tee(object):
    def __init__(self, encoding, *args):
        self.encoding = encoding
        self._streams = args
        self.errors = None

    def write(self, data):
        data = force_unicode(data, self.encoding)
        for s in self._streams:
            s.write(data)

    def writelines(self, lines):
        for line in lines:
```

```
            self.write(line)

    def flush(self):
        for s in self._streams:
            s.flush()

    def isatty(self):
        return False


class Xunit(Plugin):
    """This plugin provides test results in the standard XUnit XML format."""
    name = 'xunit'
    score = 1500
    encoding = 'UTF-8'
    error_report_file = None

    def __init__(self):
        super(Xunit, self).__init__()
        self._capture_stack = []
        self._currentStdout = None
        self._currentStderr = None

    def _timeTaken(self):
        if hasattr(self, '_timer'):
            taken = time() - self._timer
        else:
            # test died before it ran (probably error in setup())
            # or success/failure added before test started probably
            # due to custom TestResult munging
            taken = 0.0
        return taken

    def _quoteattr(self, attr):
        """Escape an XML attribute. Value can be unicode."""
        attr = xml_safe(attr)
        return saxutils.quoteattr(attr)

    def _getCls(self, id):
        if self.xunit_prefix_class:
            return self._quoteattr('%s.%s' % (self.xunit_testsuite_name, id_
↪split(id)[0]))
        else:
            return self._quoteattr(id_split(id)[0])

    def options(self, parser, env):
        """Sets additional command line options."""
        Plugin.options(self, parser, env)
        parser.add_option(
            '--xunit-file', action='store',
            dest='xunit_file', metavar="FILE",
            default=env.get('NOSE_XUNIT_FILE', 'nosetests.xml'),
            help=("Path to xml file to store the xunit report in. "
                  "Default is nosetests.xml in the working directory "
                  "[NOSE_XUNIT_FILE]"))

        parser.add_option(
            '--xunit-testsuite-name', action='store',
```

```
            dest='xunit_testsuite_name', metavar="PACKAGE",
            default=env.get('NOSE_XUNIT_TESTSUITE_NAME', 'nosetests'),
            help=("Name of the testsuite in the xunit xml, generated by plugin. "
                  "Default test suite name is nosetests."))

        parser.add_option(
            '--xunit-prefix-with-testsuite-name', action='store_true',
            dest='xunit_prefix_class',
            default=bool(env.get('NOSE_XUNIT_PREFIX_WITH_TESTSUITE_NAME', False)),
            help=("Whether to prefix the class name under test with testsuite name. "
                  "Defaults to false."))

    def configure(self, options, config):
        """Configures the xunit plugin."""
        Plugin.configure(self, options, config)
        self.config = config
        if self.enabled:
            self.stats = {'errors': 0,
                          'failures': 0,
                          'passes': 0,
                          'skipped': 0
                          }
            self.errorlist = []
            self.error_report_file_name = os.path.realpath(options.xunit_file)
            self.xunit_testsuite_name = options.xunit_testsuite_name
            self.xunit_prefix_class = options.xunit_prefix_class

    def report(self, stream):
        """Writes an Xunit-formatted XML file

        The file includes a report of test errors and failures.

        """
        self.error_report_file = codecs.open(self.error_report_file_name, 'w',
                                             self.encoding, 'replace')
        self.stats['encoding'] = self.encoding
        self.stats['testsuite_name'] = self.xunit_testsuite_name
        self.stats['total'] = (self.stats['errors'] + self.stats['failures']
                               + self.stats['passes'] + self.stats['skipped'])
        self.error_report_file.write(
            u'<?xml version="1.0" encoding="%(encoding)s"?>'
            u'<testsuite name="%(testsuite_name)s" tests="%(total)d" '
            u'errors="%(errors)d" failures="%(failures)d" '
            u'skip="%(skipped)d">' % self.stats)
        self.error_report_file.write(u''.join([force_unicode(e, self.encoding)
                                               for e in self.errorlist]))
        self.error_report_file.write(u'</testsuite>')
        self.error_report_file.close()
        if self.config.verbosity > 1:
            stream.writeln("-" * 70)
            stream.writeln("XML: %s" % self.error_report_file.name)

    def _startCapture(self):
        self._capture_stack.append((sys.stdout, sys.stderr))
        self._currentStdout = StringIO()
        self._currentStderr = StringIO()
        sys.stdout = Tee(self.encoding, self._currentStdout, sys.stdout)
        sys.stderr = Tee(self.encoding, self._currentStderr, sys.stderr)
```

```
    def startContext(self, context):
        self._startCapture()

    def stopContext(self, context):
        self._endCapture()

    def beforeTest(self, test):
        """Initializes a timer before starting a test."""
        self._timer = time()
        self._startCapture()

    def _endCapture(self):
        if self._capture_stack:
            sys.stdout, sys.stderr = self._capture_stack.pop()

    def afterTest(self, test):
        self._endCapture()
        self._currentStdout = None
        self._currentStderr = None

    def finalize(self, test):
        while self._capture_stack:
            self._endCapture()

    def _getCapturedStdout(self):
        if self._currentStdout:
            value = self._currentStdout.getvalue()
            if value:
                return '<system-out><![CDATA[%s]]></system-out>' % escape_cdata(
                       value)
        return ''

    def _getCapturedStderr(self):
        if self._currentStderr:
            value = self._currentStderr.getvalue()
            if value:
                return '<system-err><![CDATA[%s]]></system-err>' % escape_cdata(
                       value)
        return ''

    def addError(self, test, err, capt=None):
        """Add error output to Xunit report.
        """
        taken = self._timeTaken()

        if issubclass(err[0], SkipTest):
            type = 'skipped'
            self.stats['skipped'] += 1
        else:
            type = 'error'
            self.stats['errors'] += 1

        tb = format_exception(err, self.encoding)
        id = test.id()

        self.errorlist.append(
            u'<testcase classname=%(cls)s name=%(name)s time="%(taken).3f">'
```

```
            u'<%(type)s type=%(errtype)s message=%(message)s><![CDATA[%(tb)s]]>'
            u'</%(type)s>%(systemout)s%(systemerr)s</testcase>' %
            {'cls': self._getCls(id),
             'name': self._quoteattr(id_split(id)[-1]),
             'taken': taken,
             'type': type,
             'errtype': self._quoteattr(nice_classname(err[0])),
             'message': self._quoteattr(exc_message(err)),
             'tb': escape_cdata(tb),
             'systemout': self._getCapturedStdout(),
             'systemerr': self._getCapturedStderr(),
             })

    def addFailure(self, test, err, capt=None, tb_info=None):
        """Add failure output to Xunit report.
        """
        taken = self._timeTaken()
        tb = format_exception(err, self.encoding)
        self.stats['failures'] += 1
        id = test.id()

        self.errorlist.append(
            u'<testcase classname=%(cls)s name=%(name)s time="%(taken).3f">'
            u'<failure type=%(errtype)s message=%(message)s><![CDATA[%(tb)s]]>'
            u'</failure>%(systemout)s%(systemerr)s</testcase>' %
            {'cls': self._getCls(id),
             'name': self._quoteattr(id_split(id)[-1]),
             'taken': taken,
             'errtype': self._quoteattr(nice_classname(err[0])),
             'message': self._quoteattr(exc_message(err)),
             'tb': escape_cdata(tb),
             'systemout': self._getCapturedStdout(),
             'systemerr': self._getCapturedStderr(),
             })

    def addSuccess(self, test, capt=None):
        """Add success output to Xunit report.
        """
        taken = self._timeTaken()
        self.stats['passes'] += 1
        id = test.id()
        self.errorlist.append(
            '<testcase classname=%(cls)s name=%(name)s '
            'time="%(taken).3f">%(systemout)s%(systemerr)s</testcase>' %
            {'cls': self._getCls(id),
             'name': self._quoteattr(id_split(id)[-1]),
             'taken': taken,
             'systemout': self._getCapturedStdout(),
             'systemerr': self._getCapturedStderr(),
             })
```

Third-party nose plugins

Visit http://nose-plugins.jottit.com/ for a list of third-party nose plugins compatible with nose 0.9 through 0.11. If you have released a plugin that you don't see in the list, please add it!

Setuptools integration

Warning: Please note that when run under the setuptools test command, many plugins will not be available, including the builtin coverage and profiler plugins. If you want to access to all available plugins, use the *nosetests* command instead.

nose may be used with the setuptools test command. Simply specify nose.collector as the test suite in your setup file:

```
setup (
    # ...
    test_suite = 'nose.collector'
)
```

Then to find and run tests, you can run:

```
python setup.py test
```

When running under setuptools, you can configure nose settings via the environment variables detailed in the nosetests script usage message, or the setup.cfg, ~/.noserc or ~/.nose.cfg config files.

nosetests command

nose also includes its own setuptools command, `nosetests`, that provides support for all plugins and command line options. It works just like the `test` command:

```
python setup.py nosetests
```

See *nosetests setuptools command* for more information about the `nosetests` command.

Developing with nose

Get the code

nose is hosted at GitHub. You should clone this repository if you're developing a plugin or working on bug fixes for nose:

```
git clone https://github.com/nose-devs/nose
```

You should **fork** this repository if you are developing new features for nose. Then submit your changes as a pull request.

Read

Extending and customizing nose with plugins

nose has plugin hooks for loading, running, watching and reporting on tests and test runs. If you don't like the default collection scheme, or it doesn't suit the layout of your project, or you need reports in a format different from the unittest standard, or you need to collect some additional information about tests (like code coverage or profiling data), you can write a plugin to do so. See the section on *writing plugins* for more.

nose also comes with a number of built-in plugins, such as:

- Output capture

  Unless called with the `-s` (`--nocapture`) switch, nose will capture stdout during each test run, and print the captured output only for tests that fail or have errors. The captured output is printed immediately following the error or failure output for the test. (Note that output in teardown methods is captured, but can't be output with failing tests, because teardown has not yet run at the time of the failure.)

- Assert introspection

  When run with the `-d` (`--detailed-errors`) switch, nose will try to output additional information about the assert expression that failed with each failing test. Currently, this means that names in the assert expression will be expanded into any values found for them in the locals or globals in the frame in which the expression executed.

  In other words, if you have a test like:

```
def test_integers():
    a = 2
    assert a == 4, "assert 2 is 4"
```

  You will get output like:

```
File "/path/to/file.py", line XX, in test_integers:
     assert a == 4, "assert 2 is 4"
AssertionError: assert 2 is 4
  >>  assert 2 == 4, "assert 2 is 4"
```

  Please note that dotted names are not expanded, and callables are not called in the expansion.

See below for the rest of the built-in plugins.

Using Builtin plugins

See *Batteries included: builtin nose plugins*

Writing plugins

Writing Plugins

nose supports plugins for test collection, selection, observation and reporting. There are two basic rules for plugins:

- Plugin classes should subclass `nose.plugins.Plugin`.
- Plugins may implement any of the methods described in the class *IPluginInterface* in nose.plugins.base. Please note that this class is for documentary purposes only; plugins may not subclass IPluginInterface.

Hello World

Here's a basic plugin. It doesn't do much so read on for more ideas or dive into the *IPluginInterface* to see all available hooks.

```
import logging
import os

from nose.plugins import Plugin

log = logging.getLogger('nose.plugins.helloworld')

class HelloWorld(Plugin):
    name = 'helloworld'

    def options(self, parser, env=os.environ):
        super(HelloWorld, self).options(parser, env=env)

    def configure(self, options, conf):
        super(HelloWorld, self).configure(options, conf)
        if not self.enabled:
            return

    def finalize(self, result):
        log.info('Hello pluginized world!')
```

Registering

Note: Important note: the following applies only to the default plugin manager. Other plugin managers may use different means to locate and load plugins.

For nose to find a plugin, it must be part of a package that uses setuptools, and the plugin must be included in the entry points defined in the setup.py for the package:

```
setup(name='Some plugin',
    # ...
    entry_points = {
        'nose.plugins.0.10': [
            'someplugin = someplugin:SomePlugin'
            ]
        },
    # ...
    )
```

Once the package is installed with install or develop, nose will be able to load the plugin.

Registering a plugin without setuptools

It is currently possible to register a plugin programmatically by creating a custom nose runner like this :

```
import nose
from yourplugin import YourPlugin

if __name__ == '__main__':
    nose.main(addplugins=[YourPlugin()])
```

Defining options

All plugins must implement the methods `options(self, parser, env)` and `configure(self, options, conf)`. Subclasses of nose.plugins.Plugin that want the standard options should call the superclass methods.

nose uses optparse.OptionParser from the standard library to parse arguments. A plugin's `options()` method receives a parser instance. It's good form for a plugin to use that instance only to add additional arguments that take only long arguments (–like-this). Most of nose's built-in arguments get their default value from an environment variable.

A plugin's `configure()` method receives the parsed `OptionParser` options object, as well as the current config object. Plugins should configure their behavior based on the user-selected settings, and may raise exceptions if the configured behavior is nonsensical.

Logging

nose uses the logging classes from the standard library. To enable users to view debug messages easily, plugins should use `logging.getLogger()` to acquire a logger in the `nose.plugins` namespace.

Recipes

- Writing a plugin that monitors or controls test result output

  Implement any or all of `addError`, `addFailure`, etc., to monitor test results. If you also want to monitor output, implement `setOutputStream` and keep a reference to the output stream. If you want to prevent the builtin `TextTestResult` output, implement `setOutputSteam` and *return a dummy stream*. The default output will go to the dummy stream, while you send your desired output to the real stream.

  Example: examples/html_plugin/htmlplug.py

- Writing a plugin that handles exceptions

  Subclass *ErrorClassPlugin*.

  Examples: *nose.plugins.deprecated*, *nose.plugins.skip*

- Writing a plugin that adds detail to error reports

  Implement `formatError` and/or `formatFailure`. The error tuple you return (error class, error message, traceback) will replace the original error tuple.

  Examples: *nose.plugins.capture*, *nose.plugins.failuredetail*

- Writing a plugin that loads tests from files other than python modules

  Implement `wantFile` and `loadTestsFromFile`. In `wantFile`, return True for files that you want to examine for tests. In `loadTestsFromFile`, for those files, return an iterable containing TestCases (or yield them as you find them; `loadTestsFromFile` may also be a generator).

  Example: *nose.plugins.doctests*

- Writing a plugin that prints a report

  Implement `begin` if you need to perform setup before testing begins. Implement `report` and output your report to the provided stream.

  Examples: *nose.plugins.cover*, *nose.plugins.prof*

- Writing a plugin that selects or rejects tests

  Implement any or all `want*` methods. Return False to reject the test candidate, True to accept it – which means that the test candidate will pass through the rest of the system, so you must be prepared to load tests from it if tests can't be loaded by the core loader or another plugin – and None if you don't care.

  Examples: *nose.plugins.attrib*, *nose.plugins.doctests*, *nose.plugins.testid*

More Examples

See any builtin plugin or example plugin in the examples directory in the nose source distribution. There is a list of third-party plugins on jottit.

Plugin Interface

Plugin base class

class `nose.plugins.base.`**`Plugin`**

Base class for nose plugins. It's recommended but not *necessary* to subclass this class to create a plugin, but all plugins *must* implement *options(self, parser, env)* and *configure(self, options, conf)*, and must have the attributes *enabled*, *name* and *score*. The *name* attribute may contain hyphens ('-').

Plugins should not be enabled by default.

Subclassing Plugin (and calling the superclass methods in __init__, configure, and options, if you override them) will give your plugin some friendly default behavior:

- A –with-$name option will be added to the command line interface to enable the plugin, and a corresponding environment variable will be used as the default value. The plugin class's docstring will be used as the help for this option.
- The plugin will not be enabled unless this option is selected by the user.

`addOptions`(*parser*, *env=None*)

Add command-line options for this plugin.

The base plugin class adds –with-$name by default, used to enable the plugin.

> **Warning:** Don't implement addOptions unless you want to override all default option handling behavior, including warnings for conflicting options. Implement *`options`* instead.

`add_options`(*parser*, *env=None*)

Non-camel-case version of func name for backwards compatibility.

> **Warning:** DEPRECATED: Do not use this method, use *`options`* instead.

`configure`(*options*, *conf*)

Configure the plugin and system, based on selected options.

The base plugin class sets the plugin to enabled if the enable option for the plugin (self.enableOpt) is true.

`help`()

Return help for this plugin. This will be output as the help section of the –with-$name option that enables the plugin.

options (*parser*, *env*)
Register commandline options.

Implement this method for normal options behavior with protection from OptionConflictErrors. If you override this method and want the default –with-$name option to be registered, be sure to call super().

Nose plugin API

Plugins may implement any or all of the methods documented below. Please note that they *must not* subclass *IPluginInterface*; *IPluginInterface* is only a description of the plugin API.

When plugins are called, the first plugin that implements a method and returns a non-None value wins, and plugin processing ends. The exceptions to this are methods marked as *generative* or *chainable*. *generative* methods combine the output of all plugins that respond with an iterable into a single flattened iterable response (a generator, really). *chainable* methods pass the results of calling plugin A as the input to plugin B, where the positions in the chain are determined by the plugin sort order, which is in order by *score* descending.

In general, plugin methods correspond directly to methods of *nose.selector.Selector*, *nose.loader.TestLoader* and *nose.result.TextTestResult* are called by those methods when they are called. In some cases, the plugin hook doesn't neatly match the method in which it is called; for those, the documentation for the hook will tell you where in the test process it is called.

Plugin hooks fall into four broad categories: selecting and loading tests, handling errors raised by tests, preparing objects used in the testing process, and watching and reporting on test results.

Selecting and loading tests

To alter test selection behavior, implement any necessary *want** methods as outlined below. Keep in mind, though, that when your plugin returns True from a *want** method, you will send the requested object through the normal test collection process. If the object represents something from which normal tests can't be collected, you must also implement a loader method to load the tests.

Examples:

- The builtin *doctests plugin* implements *wantFile* to enable loading of doctests from files that are not python modules. It also implements *loadTestsFromModule* to load doctests from python modules, and *loadTestsFromFile* to load tests from the non-module files selected by *wantFile*.
- The builtin *attrib plugin* implements *wantFunction* and *wantMethod* so that it can reject tests that don't match the specified attributes.

Handling errors

To alter error handling behavior – for instance to catch a certain class of exception and handle it differently from the normal error or failure handling – you should subclass `nose.plugins.errorclass.ErrorClassPlugin`. See *the section on ErrorClass plugins* for more details.

Examples:

- The builtin *skip* and *deprecated* plugins are ErrorClass plugins.

Preparing test objects

To alter, get a handle on, or replace test framework objects such as the loader, result, runner, and test cases, use the appropriate prepare methods. The simplest reason to use prepare is in the case that you need to use an object yourself.

For example, the isolate plugin implements *prepareTestLoader* so that it can use the loader later on to load tests. If you return a value from a prepare method, that value will be used in place of the loader, result, runner or test case, depending on which prepare method you use. Be aware that when replacing test cases, you are replacing the *entire* test case – including the whole *run(result)* method of the *unittest.TestCase* – so if you want normal unittest test result reporting, you must implement the same calls to result as *unittest.TestCase.run*.

Examples:

- The builtin *isolate plugin* implements *prepareTestLoader* but does not replace the test loader.
- The builtin *profile plugin* implements *prepareTest* and does replace the top-level test case by returning the case wrapped in the profiler function.

Watching or reporting on tests

To record information about tests or other modules imported during the testing process, output additional reports, or entirely change test report output, implement any of the methods outlined below that correspond to TextTestResult methods.

Examples:

- The builtin *cover plugin* implements *begin* and *report* to capture and report code coverage metrics for all or selected modules loaded during testing.
- The builtin *profile plugin* implements *begin*, *prepareTest* and *report* to record and output profiling information. In this case, the plugin's *prepareTest* method constructs a function that runs the test through the hotshot profiler's runcall() method.

Plugin interface methods

class nose.plugins.base.**IPluginInterface**
IPluginInterface describes the plugin API. Do not subclass or use this class directly.

addDeprecated(*test*)
Called when a deprecated test is seen. DO NOT return a value unless you want to stop other plugins from seeing the deprecated test.

> **Warning:** DEPRECATED – check error class in addError instead

addError(*test*, *err*)
Called when a test raises an uncaught exception. DO NOT return a value unless you want to stop other plugins from seeing that the test has raised an error.

Parameters

- **test** (*nose.case.Test*) – the test case
- **err** (*3-tuple*) – sys.exc_info() tuple

addFailure(*test*, *err*)
Called when a test fails. DO NOT return a value unless you want to stop other plugins from seeing that the test has failed.

Parameters

- **test** (*nose.case.Test*) – the test case
- **err** (*sys.exc_info() tuple*) – 3-tuple

addOptions(*parser*, *env*)
Called to allow plugin to register command-line options with the parser. DO NOT return a value from this method unless you want to stop all other plugins from setting their options.

> **Warning:** DEPRECATED – implement *options* instead.

addSkip(*test*)
Called when a test is skipped. DO NOT return a value unless you want to stop other plugins from seeing the skipped test.

> **Warning:** DEPRECATED – check error class in addError instead

addSuccess(*test*)
Called when a test passes. DO NOT return a value unless you want to stop other plugins from seeing the passing test.

Parameters **test** (*nose.case.Test*) – the test case

add_options(*parser*, *env*)
Called to allow plugin to register command-line options with the parser. DO NOT return a value from this method unless you want to stop all other plugins from setting their options.

> **Warning:** DEPRECATED – implement *options* instead.

afterContext()
Called after a context (generally a module) has been lazy-loaded, imported, setup, had its tests loaded and executed, and torn down.

afterDirectory(*path*)
Called after all tests have been loaded from directory at path and run.

Parameters **path** (*string*) – the directory that has finished processing

afterImport(*filename*, *module*)
Called after module is imported from filename. afterImport is called even if the import failed.

Parameters

- **filename** (*string*) – The file that was loaded
- **module** (*string*) – The name of the module

afterTest(*test*)
Called after the test has been run and the result recorded (after stopTest).

Parameters **test** (*nose.case.Test*) – the test case

beforeContext()
Called before a context (generally a module) is examined. Because the context is not yet loaded, plugins don't get to know what the context is; so any context operations should use a stack that is pushed in *beforeContext* and popped in *afterContext* to ensure they operate symmetrically.

beforeContext and *afterContext* are mainly useful for tracking and restoring global state around possible changes from within a context, whatever the context may be. If you need to operate on contexts themselves, see *startContext* and *stopContext*, which are passed the context in question, but are called after it has been loaded (imported in the module case).

beforeDirectory(*path*)
Called before tests are loaded from directory at path.

Parameters **path** – the directory that is about to be processed

beforeImport(*filename*, *module*)
Called before module is imported from filename.

Parameters

- **filename** – The file that will be loaded
- **module** (*string*) – The name of the module found in file

beforeTest(*test*)
Called before the test is run (before startTest).

Parameters **test** (*nose.case.Test*) – the test case

begin()
Called before any tests are collected or run. Use this to perform any setup needed before testing begins.

configure(*options*, *conf*)
Called after the command line has been parsed, with the parsed options and the config container. Here, implement any config storage or changes to state or operation that are set by command line options.

DO NOT return a value from this method unless you want to stop all other plugins from being configured.

describeTest(*test*)
Return a test description.

Called by `nose.case.Test.shortDescription()`.

Parameters **test** (*nose.case.Test*) – the test case

finalize(*result*)
Called after all report output, including output from all plugins, has been sent to the stream. Use this to print final test results or perform final cleanup. Return None to allow other plugins to continue printing, or any other value to stop them.

Parameters **result** – test result object

Note: When tests are run under a test runner other than *nose.core.TextTestRunner*, such as via `python setup.py test`, this method may be called **before** the default report output is sent.

---

formatError(*test*, *err*)
Called in result.addError, before plugin.addError. If you want to replace or modify the error tuple, return a new error tuple, otherwise return err, the original error tuple.

Parameters

- **test** (*nose.case.Test*) – the test case
- **err** (*3-tuple*) – sys.exc_info() tuple

formatFailure(*test*, *err*)
Called in result.addFailure, before plugin.addFailure. If you want to replace or modify the error tuple, return a new error tuple, otherwise return err, the original error tuple.

Parameters

- **test** (*nose.case.Test*) – the test case
- **err** (*3-tuple*) – sys.exc_info() tuple

handleError(*test*, *err*)
Called on addError. To handle the error yourself and prevent normal error processing, return a true value.

Parameters

- **test** (*nose.case.Test*) – the test case
- **err** (*3-tuple*) – sys.exc_info() tuple

handleFailure(*test*, *err*)
Called on addFailure. To handle the failure yourself and prevent normal failure processing, return a true value.

Parameters

- **test** (*nose.case.Test*) – the test case
- **err** (*3-tuple*) – sys.exc_info() tuple

loadTestsFromDir(*path*)
Return iterable of tests from a directory. May be a generator. Each item returned must be a runnable unittest.TestCase (or subclass) instance or suite instance. Return None if your plugin cannot collect any tests from directory.

Parameters **path** – The path to the directory.

loadTestsFromFile(*filename*)
Return tests in this file. Return None if you are not interested in loading any tests, or an iterable if you are and can load some. May be a generator. *If you are interested in loading tests from the file and encounter no errors, but find no tests, yield False or return [False].*

Note: This method replaces loadTestsFromPath from the 0.9 API.

Parameters **filename** – The full path to the file or directory.

loadTestsFromModule(*module*, *path=None*)
Return iterable of tests in a module. May be a generator. Each item returned must be a runnable unittest.TestCase (or subclass) instance. Return None if your plugin cannot collect any tests from module.

Parameters

- **module** (*python module*) – The module object
- **path** – the path of the module to search, to distinguish from namespace package modules

---

Note: NEW. The `path` parameter will only be passed by nose 0.11 or above.

---

loadTestsFromName(*name*, *module=None*, *importPath=None*)
Return tests in this file or module. Return None if you are not able to load any tests, or an iterable if you are. May be a generator.

Parameters

- **name** – The test name. May be a file or module name plus a test callable. Use split_test_name to split into parts. Or it might be some crazy name of your own devising, in which case, do whatever you want.
- **module** – Module from which the name is to be loaded

- **importPath** – Path from which file (must be a python module) was found

> **Warning:** DEPRECATED: this argument will NOT be passed.

loadTestsFromNames(*names*, *module=None*)
Return a tuple of (tests loaded, remaining names). Return None if you are not able to load any tests. Multiple plugins may implement loadTestsFromNames; the remaining name list from each will be passed to the next as input.

Parameters

- **names** (*iterable*) – List of test names.
- **module** – Module from which the names are to be loaded

loadTestsFromPath(*path*)

> **Warning:** DEPRECATED – use loadTestsFromFile instead

loadTestsFromTestCase(*cls*)
Return tests in this test case class. Return None if you are not able to load any tests, or an iterable if you are. May be a generator.

Parameters **cls** – The test case class. Must be subclass of `unittest.TestCase`.

loadTestsFromTestClass(*cls*)
Return tests in this test class. Class will *not* be a unittest.TestCase subclass. Return None if you are not able to load any tests, an iterable if you are. May be a generator.

Parameters **cls** – The test case class. Must be **not** be subclass of `unittest.TestCase`.

makeTest(*obj*, *parent*)
Given an object and its parent, return or yield one or more test cases. Each test must be a unittest.TestCase (or subclass) instance. This is called before default test loading to allow plugins to load an alternate test case or cases for an object. May be a generator.

Parameters

- **obj** – The object to be made into a test
- **parent** – The parent of obj (eg, for a method, the class)

options(*parser*, *env*)
Called to allow plugin to register command line options with the parser.

DO NOT return a value from this method unless you want to stop all other plugins from setting their options.

Parameters

- **parser** (`ConfigParser.ConfigParser`) – options parser instance
- **env** – environment, default is os.environ

prepareTest(*test*)
Called before the test is run by the test runner. Please note the article *the* in the previous sentence: prepareTest is called *only once*, and is passed the test case or test suite that the test runner will execute. It is *not* called for each individual test case. If you return a non-None value, that return value will be run as

the test. Use this hook to wrap or decorate the test with another function. If you need to modify or wrap individual test cases, use *prepareTestCase* instead.

Parameters **test** (*nose.case.Test*) – the test case

prepareTestCase(*test*)
Prepare or wrap an individual test case. Called before execution of the test. The test passed here is a nose.case.Test instance; the case to be executed is in the test attribute of the passed case. To modify the test to be run, you should return a callable that takes one argument (the test result object) – it is recommended that you *do not* side-effect the nose.case.Test instance you have been passed.

Keep in mind that when you replace the test callable you are replacing the run() method of the test case – including the exception handling and result calls, etc.

Parameters **test** (*nose.case.Test*) – the test case

prepareTestLoader(*loader*)
Called before tests are loaded. To replace the test loader, return a test loader. To allow other plugins to process the test loader, return None. Only one plugin may replace the test loader. Only valid when using nose.TestProgram.

Parameters **loader** – *nose.loader.TestLoader* (or other loader) instance

prepareTestResult(*result*)
Called before the first test is run. To use a different test result handler for all tests than the given result, return a test result handler. NOTE however that this handler will only be seen by tests, that is, inside of the result proxy system. The TestRunner and TestProgram – whether nose's or other – will continue to see the original result handler. For this reason, it is usually better to monkeypatch the result (for instance, if you want to handle some exceptions in a unique way). Only one plugin may replace the result, but many may monkeypatch it. If you want to monkeypatch and stop other plugins from doing so, monkeypatch and return the patched result.

Parameters **result** – *nose.result.TextTestResult* (or other result) instance

prepareTestRunner(*runner*)
Called before tests are run. To replace the test runner, return a test runner. To allow other plugins to process the test runner, return None. Only valid when using nose.TestProgram.

Parameters **runner** – *nose.core.TextTestRunner* (or other runner) instance

report(*stream*)
Called after all error output has been printed. Print your plugin's report to the provided stream. Return None to allow other plugins to print reports, any other value to stop them.

Parameters **stream** (*file-like object*) – stream object; send your output here

setOutputStream(*stream*)
Called before test output begins. To direct test output to a new stream, return a stream object, which must implement a *write(msg)* method. If you only want to note the stream, not capture or redirect it, then return None.

Parameters **stream** (*file-like object*) – stream object; send your output here

startContext(*context*)
Called before context setup and the running of tests in the context. Note that tests have already been *loaded* from the context before this call.

Parameters **context** – the context about to be setup. May be a module or class, or any other object that contains tests.

startTest(*test*)
Called before each test is run. DO NOT return a value unless you want to stop other plugins from seeing the test start.

Parameters **test** (*nose.case.Test*) – the test case

stopContext(*context*)
Called after the tests in a context have run and the context has been torn down.

Parameters **context** – the context that has been torn down. May be a module or class, or any other object that contains tests.

stopTest(*test*)
Called after each test is run. DO NOT return a value unless you want to stop other plugins from seeing that the test has stopped.

Parameters **test** (*nose.case.Test*) – the test case

testName(*test*)
Return a short test name. Called by *nose.case.Test.__str__*.

Parameters **test** (*nose.case.Test*) – the test case

wantClass(*cls*)
Return true if you want the main test selector to collect tests from this class, false if you don't, and None if you don't care.

Parameters **cls** – The class being examined by the selector

wantDirectory(*dirname*)
Return true if you want test collection to descend into this directory, false if you do not, and None if you don't care.

Parameters **dirname** – Full path to directory being examined by the selector

wantFile(*file*)
Return true if you want to collect tests from this file, false if you do not and None if you don't care.

Change from 0.9: The optional package parameter is no longer passed.

Parameters **file** – Full path to file being examined by the selector

wantFunction(*function*)
Return true to collect this function as a test, false to prevent it from being collected, and None if you don't care.

Parameters **function** – The function object being examined by the selector

wantMethod(*method*)
Return true to collect this method as a test, false to prevent it from being collected, and None if you don't care.

Parameters **method** (*unbound method*) – The method object being examined by the selector

wantModule(*module*)
Return true if you want to collection to descend into this module, false to prevent the collector from descending into the module, and None if you don't care.

Parameters **module** (*python module*) – The module object being examined by the selector

wantModuleTests(*module*)

> **Warning:** DEPRECATED – this method will not be called, it has been folded into wantModule.

ErrorClass Plugins

ErrorClass plugins provide an easy way to add support for custom handling of particular classes of exceptions.

An ErrorClass plugin defines one or more ErrorClasses and how each is handled and reported on. Each error class is stored in a different attribute on the result, and reported separately. Each error class must indicate the exceptions that fall under that class, the label to use for reporting, and whether exceptions of the class should be considered as failures for the whole test run.

ErrorClasses use a declarative syntax. Assign an ErrorClass to the attribute you wish to add to the result object, defining the exceptions, label and isfailure attributes. For example, to declare an ErrorClassPlugin that defines TodoErrors (and subclasses of TodoError) as an error class with the label 'TODO' that is considered a failure, do this:

```
>>> class Todo(Exception):
...     pass
>>> class TodoError(ErrorClassPlugin):
...     todo = ErrorClass(Todo, label='TODO', isfailure=True)
```

The MetaErrorClass metaclass translates the ErrorClass declarations into the tuples used by the error handling and reporting functions in the result. This is an internal format and subject to change; you should always use the declarative syntax for attaching ErrorClasses to an ErrorClass plugin.

```
>>> TodoError.errorClasses
((<class ...Todo...>, ('todo', 'TODO', True)),)
```

Let's see the plugin in action. First some boilerplate.

```
>>> import sys
>>> import unittest
>>> try:
...     # 2.7+
...     from unittest.runner import _WritelnDecorator
... except ImportError:
...     from unittest import _WritelnDecorator
...
>>> buf = _WritelnDecorator(sys.stdout)
```

Now define a test case that raises a Todo.

```
>>> class TestTodo(unittest.TestCase):
...     def runTest(self):
...         raise Todo("I need to test something")
>>> case = TestTodo()
```

Prepare the result using our plugin. Normally this happens during the course of test execution within nose – you won't be doing this yourself. For the purposes of this testing document, I'm stepping through the internal process of nose so you can see what happens at each step.

```
>>> plugin = TodoError()
>>> from nose.result import _TextTestResult
>>> result = _TextTestResult(stream=buf, descriptions=0, verbosity=2)
>>> plugin.prepareTestResult(result)
```

Now run the test. TODO is printed.

```
>>> _ = case(result)
runTest (....TestTodo) ... TODO: I need to test something
```

Errors and failures are empty, but todo has our test:

```
>>> result.errors
[]
>>> result.failures
[]
>>> result.todo
[(<....TestTodo testMethod=runTest>, '...Todo: I need to test something\n')]
>>> result.printErrors()

======================================================================
TODO: runTest (....TestTodo)
----------------------------------------------------------------------
Traceback (most recent call last):
...
...Todo: I need to test something
```

Since we defined a Todo as a failure, the run was not successful.

```
>>> result.wasSuccessful()
False
```

Error class methods

class nose.plugins.errorclass.**ErrorClassPlugin**
Base class for ErrorClass plugins. Subclass this class and declare the exceptions that you wish to handle as attributes of the subclass.

Documenting plugins

A parable. If a plugin is released on pypi without any documentation, does anyone care?

To make it easy to document your plugins, nose includes a Sphinx extension that will automatically generate plugin docs like those for nose's builtin plugins. Simply add 'nose.sphinx.pluginopts' to the list of extensions in your conf.py:

```
extensions = ['sphinx.ext.autodoc', 'sphinx.ext.intersphinx',
              'nose.sphinx.pluginopts']
```

Then in your plugin documents, include a title and the `.. autoplugin` directive:

```
My Cool Plugin
==============

.. autoplugin :: package.module.with.plugin
   :plugin: PluginClass
```

The `:plugin:` option is optional. In most cases, the directive will automatically detect which class in the named module is the plugin to be documented.

The output of the directive includes the docstring of the plugin module, the options defined by the plugin, autodoc generated for the plugin class, and the plugin module source. This is roughly equivalent to:

```
My Cool Plugin
==============

.. automodule :: package.module.with.plugin

Options
-------

.. cmdoption :: --with-coolness

   Help text of the coolness option.

.. cmdoption ::

Plugin
-------

.. autoclass :: package.module.with.plugin.PluginClass
   :members:

Source
------

.. include :: path/to/package/module/with/plugin.py
   :literal:
```

Document your plugins! Your users might not thank you – but at least you'll *have* some users.

Testing plugins

Testing Plugins

The plugin interface is well-tested enough to safely unit test your use of its hooks with some level of confidence. However, there is also a mixin for unittest.TestCase called PluginTester that's designed to test plugins in their native runtime environment.

Here's a simple example with a do-nothing plugin and a composed suite.

```
>>> import unittest
>>> from nose.plugins import Plugin, PluginTester
>>> class FooPlugin(Plugin):
...     pass
>>> class TestPluginFoo(PluginTester, unittest.TestCase):
...     activate = '--with-foo'
...     plugins = [FooPlugin()]
...     def test_foo(self):
...         for line in self.output:
...             # i.e. check for patterns
...             pass
...
...         # or check for a line containing ...
...         assert "ValueError" in self.output
...     def makeSuite(self):
...         class TC(unittest.TestCase):
...             def runTest(self):
...                 raise ValueError("I hate foo")
```

```
...         return [TC('runTest')]
...
>>> res = unittest.TestResult()
>>> case = TestPluginFoo('test_foo')
>>> _ = case(res)
>>> res.errors
[]
>>> res.failures
[]
>>> res.wasSuccessful()
True
>>> res.testsRun
1
```

And here is a more complex example of testing a plugin that has extra arguments and reads environment variables.

```
>>> import unittest, os
>>> from nose.plugins import Plugin, PluginTester
>>> class FancyOutputter(Plugin):
...     name = "fancy"
...     def configure(self, options, conf):
...         Plugin.configure(self, options, conf)
...         if not self.enabled:
...             return
...         self.fanciness = 1
...         if options.more_fancy:
...             self.fanciness = 2
...         if 'EVEN_FANCIER' in self.env:
...             self.fanciness = 3
...
...     def options(self, parser, env=os.environ):
...         self.env = env
...         parser.add_option('--more-fancy', action='store_true')
...         Plugin.options(self, parser, env=env)
...
...     def report(self, stream):
...         stream.write("FANCY " * self.fanciness)
...
>>> class TestFancyOutputter(PluginTester, unittest.TestCase):
...     activate = '--with-fancy' # enables the plugin
...     plugins = [FancyOutputter()]
...     args = ['--more-fancy']
...     env = {'EVEN_FANCIER': '1'}
...
...     def test_fancy_output(self):
...         assert "FANCY FANCY FANCY" in self.output, (
...                                         "got: %s" % self.output)
...     def makeSuite(self):
...         class TC(unittest.TestCase):
...             def runTest(self):
...                 raise ValueError("I hate fancy stuff")
...         return [TC('runTest')]
...
>>> res = unittest.TestResult()
>>> case = TestFancyOutputter('test_fancy_output')
>>> _ = case(res)
>>> res.errors
[]
```

```
>>> res.failures
[]
>>> res.wasSuccessful()
True
>>> res.testsRun
1
```

PluginTester methods

class nose.plugins.plugintest.**PluginTester**

A mixin for testing nose plugins in their runtime environment.

Subclass this and mix in unittest.TestCase to run integration/functional tests on your plugin. When setUp() is called, the stub test suite is executed with your plugin so that during an actual test you can inspect the artifacts of how your plugin interacted with the stub test suite.

- activate
  - the argument to send nosetests to activate the plugin
- suitepath
  - if set, this is the path of the suite to test. Otherwise, you will need to use the hook, makeSuite()
- plugins
  - the list of plugins to make available during the run. Note that this does not mean these plugins will be *enabled* during the run – only the plugins enabled by the activate argument or other settings in argv or env will be enabled.
- args
  - a list of arguments to add to the nosetests command, in addition to the activate argument
- env
  - optional dict of environment variables to send nosetests

makeSuite()

returns a suite object of tests to run (unittest.TestSuite())

If self.suitepath is None, this must be implemented. The returned suite object will be executed with all plugins activated. It may return None.

Here is an example of a basic suite object you can return

```
>>> import unittest
>>> class SomeTest(unittest.TestCase):
...     def runTest(self):
...         raise ValueError("Now do something, plugin!")
...
>>> unittest.TestSuite([SomeTest()])
<unittest...TestSuite tests=[<...SomeTest testMethod=runTest>]>
```

setUp()

runs nosetests with the specified test suite, all plugins activated.

nose internals

Test runner and main()

Implements nose test program and collector.

class nose.core.**TestProgram**(*module=None*, *defaultTest='.'*, *argv=None*, *testRunner=None*, *testLoader=None*, *env=None*, *config=None*, *suite=None*, *exit=True*, *plugins=None*, *addplugins=None*)

Collect and run tests, returning success or failure.

The arguments to TestProgram() are the same as to *main()* and *run()*:

- module: All tests are in this module (default: None)
- defaultTest: Tests to load (default: '.')
- argv: Command line arguments (default: None; sys.argv is read)
- testRunner: Test runner instance (default: None)
- testLoader: Test loader instance (default: None)
- env: Environment; ignored if config is provided (default: None; os.environ is read)
- config: *nose.config.Config* instance (default: None)
- suite: Suite or list of tests to run (default: None). Passing a suite or lists of tests will bypass all test discovery and loading. *ALSO NOTE* that if you pass a unittest.TestSuite instance as the suite, context fixtures at the class, module and package level will not be used, and many plugin hooks will not be called. If you want normal nose behavior, either pass a list of tests, or a fully-configured *nose.suite.ContextSuite*.
- exit: Exit after running tests and printing report (default: True)
- plugins: List of plugins to use; ignored if config is provided (default: load plugins with DefaultPluginManager)
- addplugins: List of **extra** plugins to use. Pass a list of plugin instances in this argument to make custom plugins available while still using the DefaultPluginManager.

createTests()
Create the tests to run. If a self.suite is set, then that suite will be used. Otherwise, tests will be loaded from the given test names (self.testNames) using the test loader.

makeConfig(*env*, *plugins=None*)
Load a Config, pre-filled with user config files if any are found.

parseArgs(*argv*)
Parse argv and env and configure running environment.

runTests()
Run Tests. Returns true on success, false on failure, and sets self.success to the same value.

showPlugins()
Print list of available plugins.

nose.core.**main**
alias of *TestProgram*

nose.core.**run**(**arg*, ***kw*)
Collect and run tests, returning success or failure.

The arguments to *run()* are the same as to *main()*:

- module: All tests are in this module (default: None)
- defaultTest: Tests to load (default: '.')
- argv: Command line arguments (default: None; sys.argv is read)
- testRunner: Test runner instance (default: None)
- testLoader: Test loader instance (default: None)
- env: Environment; ignored if config is provided (default: None; os.environ is read)
- config: *nose.config.Config* instance (default: None)
- suite: Suite or list of tests to run (default: None). Passing a suite or lists of tests will bypass all test discovery and loading. *ALSO NOTE* that if you pass a unittest.TestSuite instance as the suite, context fixtures at the class, module and package level will not be used, and many plugin hooks will not be called. If you want normal nose behavior, either pass a list of tests, or a fully-configured *nose.suite.ContextSuite*.
- plugins: List of plugins to use; ignored if config is provided (default: load plugins with DefaultPluginManager)
- addplugins: List of **extra** plugins to use. Pass a list of plugin instances in this argument to make custom plugins available while still using the DefaultPluginManager.

With the exception that the `exit` argument is always set to False.

`nose.core.`**`run_exit`**
alias of *TestProgram*

`nose.core.`**`runmodule`**(*name='__main__'*, ***kw*)
Collect and run tests in a single module only. Defaults to running tests in __main__. Additional arguments to TestProgram may be passed as keyword arguments.

`nose.core.`**`collector`**()
TestSuite replacement entry point. Use anywhere you might use a unittest.TestSuite. The collector will, by default, load options from all config files and execute loader.loadTestsFromNames() on the configured testNames, or '.' if no testNames are configured.

class `nose.core.`**`TextTestRunner`**(*stream=<open file '<stderr>', mode 'w'>*, *descriptions=1*, *verbosity=1*, *config=None*)
Test runner that uses nose's TextTestResult to enable errorClasses, as well as providing hooks for plugins to override or replace the test output stream, results, and the test case itself.

`run`(*test*)
Overrides to provide plugin hooks and defer all output to the test result class.

Test Loader

nose's test loader implements the same basic functionality as its superclass, unittest.TestLoader, but extends it by more liberal interpretations of what may be a test and how a test may be named.

class `nose.loader.`**`TestLoader`**(*config=None*, *importer=None*, *workingDir=None*, *selector=None*)
Test loader that extends unittest.TestLoader to:

- Load tests from test-like functions and classes that are not unittest.TestCase subclasses
- Find and load test modules in a directory
- Support tests that are generators
- Support easy extensions of or changes to that behavior through plugins

getTestCaseNames(*testCaseClass*)
Override to select with selector, unless config.getTestCaseNamesCompat is True

loadTestsFromDir(*path*)
Load tests from the directory at path. This is a generator – each suite of tests from a module or other file is yielded and is expected to be executed before the next file is examined.

loadTestsFromFile(*filename*)
Load tests from a non-module file. Default is to raise a ValueError; plugins may implement *loadTestsFromFile* to provide a list of tests loaded from the file.

loadTestsFromGenerator(*generator*, *module*)
Lazy-load tests from a generator function. The generator function may yield either:

- a callable, or
- a function name resolvable within the same module

loadTestsFromGeneratorMethod(*generator*, *cls*)
Lazy-load tests from a generator method.

This is more complicated than loading from a generator function, since a generator method may yield:

- a function
- a bound or unbound method, or
- a method name

loadTestsFromModule(*module*, *path=None*, *discovered=False*)
Load all tests from module and return a suite containing them. If the module has been discovered and is not test-like, the suite will be empty by default, though plugins may add their own tests.

loadTestsFromName(*name*, *module=None*, *discovered=False*)
Load tests from the entity with the given name.

The name may indicate a file, directory, module, or any object within a module. See *nose.util.split_test_name* for details on test name parsing.

loadTestsFromNames(*names*, *module=None*)
Load tests from all names, returning a suite containing all tests.

loadTestsFromTestCase(*testCaseClass*)
Load tests from a unittest.TestCase subclass.

loadTestsFromTestClass(*cls*)
Load tests from a test class that is *not* a unittest.TestCase subclass.

In this case, we can't depend on the class's *__init__* taking method name arguments, so we have to compose a MethodTestCase for each method in the class that looks testlike.

parseGeneratedTest(*test*)
Given the yield value of a test generator, return a func and args.

This is used in the two loadTestsFromGenerator* methods.

resolve(*name*, *module*)
Resolve name within module

nose.loader.**defaultTestLoader**
alias of *TestLoader*

Test Selection

Test selection is handled by a Selector. The test loader calls the appropriate selector method for each object it encounters that it thinks may be a test.

class nose.selector.**Selector**(*config*)
Core test selector. Examines test candidates and determines whether, given the specified configuration, the test candidate should be selected as a test.

matches(*name*)
Does the name match my requirements?

To match, a name must match config.testMatch OR config.include and it must not match config.exclude

wantClass(*cls*)
Is the class a wanted test class?

A class must be a unittest.TestCase subclass, or match test name requirements. Classes that start with _ are always excluded.

wantDirectory(*dirname*)
Is the directory a wanted test directory?

All package directories match, so long as they do not match exclude. All other directories must match test requirements.

wantFile(*file*)
Is the file a wanted test file?

The file must be a python source file and match testMatch or include, and not match exclude. Files that match ignore are *never* wanted, regardless of plugin, testMatch, include or exclude settings.

wantFunction(*function*)
Is the function a test function?

wantMethod(*method*)
Is the method a test method?

wantModule(*module*)
Is the module a test module?

The tail of the module name must match test requirements. One exception: we always want __main__.

nose.selector.**defaultSelector**
alias of *Selector*

class nose.selector.**TestAddress**(*name*, *workingDir=None*)
A test address represents a user's request to run a particular test. The user may specify a filename or module (or neither), and/or a callable (a class, function, or method). The naming format for test addresses is:

filename_or_module:callable

Filenames that are not absolute will be made absolute relative to the working dir.

The filename or module part will be considered a module name if it doesn't look like a file, that is, if it doesn't exist on the file system and it doesn't contain any directory separators and it doesn't end in .py.

Callables may be a class name, function name, method name, or class.method specification.

Configuration

class nose.config.**Config**(***kw*)

nose configuration.

Instances of Config are used throughout nose to configure behavior, including plugin lists. Here are the default values for all config keys:

```
self.env = env = kw.pop('env', {})
self.args = ()
self.testMatch = re.compile(env.get('NOSE_TESTMATCH', r'(?:|_)[Tt]est'))
self.addPaths = not env.get('NOSE_NOPATH', False)
self.configSection = 'nosetests'
self.debug = env.get('NOSE_DEBUG')
self.debugLog = env.get('NOSE_DEBUG_LOG')
self.exclude = None
self.getTestCaseNamesCompat = False
self.includeExe = env.get('NOSE_INCLUDE_EXE',
                          sys.platform in exe_allowed_platforms)
self.ignoreFiles = (re.compile(r'^\.'),
                    re.compile(r'^_'),
                    re.compile(r'^setup\.py$')
                    )
self.include = None
self.loggingConfig = None
self.logStream = sys.stderr
self.options = NoOptions()
self.parser = None
self.plugins = NoPlugins()
self.srcDirs = ('lib', 'src')
self.runOnInit = True
self.stopOnError = env.get('NOSE_STOP', False)
self.stream = sys.stderr
self.testNames = ()
self.verbosity = int(env.get('NOSE_VERBOSE', 1))
self.where = ()
self.py3where = ()
self.workingDir = None
```

configure(*argv=None*, *doc=None*)

Configure the nose running environment. Execute configure before collecting tests with nose.TestCollector to enable output capture and other features.

configureLogging()

Configure logging for nose, or optionally other packages. Any logger name may be set with the debug option, and that logger will be set to debug level and be assigned the same handler as the nose loggers, unless it already has a handler.

configureWhere(*where*)

Configure the working directory or directories for the test run.

default()

Reset all config values to defaults.

getParser(*doc=None*)

Get the command line option parser.

help(*doc=None*)

Return the generated help message

class nose.config.**ConfiguredDefaultsOptionParser**(*parser*, *config_section*, *error=None*, *file_error=None*)
Handler for options from commandline and config files.

class nose.config.**NoOptions**
Options container that returns None for all options.

nose.config.**all_config_files**()
Return path to any existing user config files, plus any setup.cfg in the current working directory.

nose.config.**flag**(*val*)
Does the value look like an on/off flag?

nose.config.**user_config_files**()
Return path to any existing user config files

Test Cases

nose unittest.TestCase subclasses. It is not necessary to subclass these classes when writing tests; they are used internally by nose.loader.TestLoader to create test cases from test functions and methods in test classes.

class nose.case.**Test**(*test*, *config=None*, *resultProxy=None*)
The universal test case wrapper.

When a plugin sees a test, it will always see an instance of this class. To access the actual test case that will be run, access the test property of the nose.case.Test instance.

address()
Return a round-trip name for this test, a name that can be fed back as input to loadTestByName and (assuming the same plugin configuration) result in the loading of this test.

afterTest(*result*)
Called after test is complete (after result.stopTest)

beforeTest(*result*)
Called before test is run (before result.startTest)

context
Get the context object of this test (if any).

exc_info()
Extract exception info.

id()
Get a short(er) description of the test

run(*result*)
Modified run for the test wrapper.

From here we don't call result.startTest or stopTest or addSuccess. The wrapper calls addError/addFailure only if its own setup or teardown fails, or running the wrapped test fails (eg, if the wrapped "test" is not callable).

Two additional methods are called, beforeTest and afterTest. These give plugins a chance to modify the wrapped test before it is called and do cleanup after it is called. They are called unconditionally.

runTest(*result*)
Run the test. Plugins may alter the test by returning a value from prepareTestCase. The value must be callable and must accept one argument, the result instance.

class nose.failure.**Failure**(*exc_class*, *exc_val*, *tb=None*, *address=None*)
Unloadable or unexecutable test.

A Failure case is placed in a test suite to indicate the presence of a test that could not be loaded or executed. A common example is a test module that fails to import.

Test Suites

Provides a LazySuite, which is a suite whose test list is a generator function, and ContextSuite,which can run fixtures (setup/teardown functions or methods) for the context that contains its tests.

class nose.suite.**ContextList**(*tests*, *context=None*)
Not quite a suite – a group of tests in a context. This is used to hint the ContextSuiteFactory about what context the tests belong to, in cases where it may be ambiguous or missing.

class nose.suite.**ContextSuite**(*tests=()*, *context=None*, *factory=None*, *config=None*, *resultProxy=None*, *can_split=True*)
A suite with context.

A ContextSuite executes fixtures (setup and teardown functions or methods) for the context containing its tests.

The context may be explicitly passed. If it is not, a context (or nested set of contexts) will be constructed by examining the tests in the suite.

exc_info()
Hook for replacing error tuple output

failureException
alias of AssertionError

run(*result*)
Run tests in suite inside of suite fixtures.

class nose.suite.**ContextSuiteFactory**(*config=None*, *suiteClass=None*, *resultProxy=<object object>*)
Factory for ContextSuites. Called with a collection of tests, the factory decides on a hierarchy of contexts by introspecting the collection or the tests themselves to find the objects containing the test objects. It always returns one suite, but that suite may consist of a hierarchy of nested suites.

ancestry(*context*)
Return the ancestry of the context (that is, all of the packages and modules containing the context), in order of descent with the outermost ancestor last. This method is a generator.

mixedSuites(*tests*)
The complex case where there are tests that don't all share the same context. Groups tests into suites with common ancestors, according to the following (essentially tail-recursive) procedure:

Starting with the context of the first test, if it is not None, look for tests in the remaining tests that share that ancestor. If any are found, group into a suite with that ancestor as the context, and replace the current suite with that suite. Continue this process for each ancestor of the first test, until all ancestors have been processed. At this point if any tests remain, recurse with those tests as the input, returning a list of the common suite (which may be the suite or test we started with, if no common tests were found) plus the results of recursion.

suiteClass
alias of *ContextSuite*

class nose.suite.**FinalizingSuiteWrapper**(*suite*, *finalize*)
Wraps suite and calls final function after suite has executed. Used to call final functions in cases (like running in the standard test runner) where test running is not under nose's control.

class nose.suite.**LazySuite**(*tests=()*)
A suite that may use a generator as its list of tests

exception nose.suite.**MixedContextError**
Error raised when a context suite sees tests from more than one context.

Test Result

Provides a TextTestResult that extends unittest's _TextTestResult to provide support for error classes (such as the builtin skip and deprecated classes), and hooks for plugins to take over or extend reporting.

class nose.result.**TextTestResult**(*stream*, *descriptions*, *verbosity*, *config=None*, *errorClasses=None*)
Text test result that extends unittest's default test result support for a configurable set of errorClasses (eg, Skip, Deprecated, TODO) that extend the errors/failures/success triad.

addError(*test*, *err*)
Overrides normal addError to add support for errorClasses. If the exception is a registered class, the error will be added to the list for that class, not errors.

printErrors()
Overrides to print all errorClasses errors as well.

printSummary(*start*, *stop*)
Called by the test runner to print the final summary of test run results.

wasSuccessful()
Overrides to check that there are no errors in errorClasses lists that are marked as errors and should cause a run to fail.

Result Proxy

The result proxy wraps the result instance given to each test. It performs two functions: enabling extended error/failure reporting and calling plugins.

As each result event is fired, plugins are called with the same event; however, plugins are called with the nose.case.Test instance that wraps the actual test. So when a test fails and calls result.addFailure(self, err), the result proxy calls addFailure(self.test, err) for each plugin. This allows plugins to have a single stable interface for all test types, and also to manipulate the test object itself by setting the *test* attribute of the nose.case.Test that they receive.

class nose.proxy.**ResultProxy**(*result*, *test*, *config=None*)
Proxy to TestResults (or other results handler).

One ResultProxy is created for each nose.case.Test. The result proxy calls plugins with the nose.case.Test instance (instead of the wrapped test case) as each result call is made. Finally, the real result method is called, also with the nose.case.Test instance as the test parameter.

errors
Tests that raised an exception

failures
Tests that failed

shouldStop
Should the test run stop?

testsRun
Number of tests run

class nose.proxy.**ResultProxyFactory**(*config=None*)
Factory for result proxies. Generates a ResultProxy bound to each test and the result passed to the test.

nose.proxy.**proxied_attribute**(*local_attr*, *proxied_attr*, *doc*)
Create a property that proxies attribute proxied_attr through the local attribute local_attr.

Plugin Manager

A plugin manager class is used to load plugins, manage the list of loaded plugins, and proxy calls to those plugins.

The plugin managers provided with nose are:

PluginManager This manager doesn't implement loadPlugins, so it can only work with a static list of plugins.

BuiltinPluginManager This manager loads plugins referenced in nose.plugins.builtin.

EntryPointPluginManager This manager uses setuptools entrypoints to load plugins.

ExtraPluginsPluginManager This manager loads extra plugins specified with the keyword *addplugins*.

DefaultPluginMananger This is the manager class that will be used by default. If setuptools is installed, it is a subclass of *EntryPointPluginManager* and *BuiltinPluginManager*; otherwise, an alias to *BuiltinPluginManager*.

RestrictedPluginManager This manager is for use in test runs where some plugin calls are not available, such as runs started with python setup.py test, where the test runner is the default unittest TextTestRunner. It is a subclass of DefaultPluginManager.

Writing a plugin manager

If you want to load plugins via some other means, you can write a plugin manager and pass an instance of your plugin manager class when instantiating the *nose.config.Config* instance that you pass to TestProgram (or main() or run()).

To implement your plugin loading scheme, implement loadPlugins(), and in that method, call addPlugin() with an instance of each plugin you wish to make available. Make sure to call super(self).loadPlugins() as well if have subclassed a manager other than PluginManager.

class nose.plugins.manager.**PluginManager**(*plugins=()*, *proxyClass=None*)
Base class for plugin managers. PluginManager is intended to be used only with a static list of plugins. The loadPlugins() implementation only reloads plugins from _extraplugins to prevent those from being overridden by a subclass.

The basic functionality of a plugin manager is to proxy all unknown attributes through a PluginProxy to a list of plugins.

Note that the list of plugins *may not* be changed after the first plugin call.

addPlugins(*plugins=()*, *extraplugins=()*)
extraplugins are maintained in a separate list and re-added by loadPlugins() to prevent their being overwritten by plugins added by a subclass of PluginManager

configure(*options*, *config*)
Configure the set of plugins with the given options and config instance. After configuration, disabled plugins are removed from the plugins list.

plugins
Access the list of plugins managed by this plugin manager

proxyClass
alias of PluginProxy

class nose.plugins.manager.**EntryPointPluginManager**(*plugins=()*, *proxyClass=None*)
Plugin manager that loads plugins from the *nose.plugins* and *nose.plugins.0.10* entry points.

loadPlugins()
Load plugins by iterating the *nose.plugins* entry point.

class nose.plugins.manager.**BuiltinPluginManager**(*plugins=()*, *proxyClass=None*)
Plugin manager that loads plugins from the list in *nose.plugins.builtin*.

loadPlugins()
Load plugins in nose.plugins.builtin

class nose.plugins.manager.**RestrictedPluginManager**(*plugins=()*, *exclude=()*, *load=True*)
Plugin manager that restricts the plugin list to those not excluded by a list of exclude methods. Any plugin that implements an excluded method will be removed from the manager's plugin list after plugins are loaded.

Importer

Implements an importer that looks only in specific path (ignoring sys.path), and uses a per-path cache in addition to sys.modules. This is necessary because test modules in different directories frequently have the same names, which means that the first loaded would mask the rest when using the builtin importer.

class nose.importer.**Importer**(*config=None*)
An importer class that does only path-specific imports. That is, the given module is not searched for on sys.path, but only at the path or in the directory specified.

importFromDir(*dir*, *fqname*)
Import a module *only* from path, ignoring sys.path and reloading if the version in sys.modules is not the one we want.

importFromPath(*path*, *fqname*)
Import a dotted-name package whose tail is at path. In other words, given foo.bar and path/to/foo/bar.py, import foo from path/to/foo then bar from path/to/foo/bar, returning bar.

nose.importer.**add_path**(*path*, *config=None*)
Ensure that the path, or the root of the current package (if path is in a package), is in sys.path.

nosetests setuptools command

The easiest way to run tests with nose is to use the *nosetests* setuptools command:

```
python setup.py nosetests
```

This command has one *major* benefit over the standard *test* command: *all nose plugins are supported*.

To configure the *nosetests* command, add a [nosetests] section to your setup.cfg. The [nosetests] section can contain any command line arguments that nosetests supports. The differences between issuing an option on the command line and adding it to setup.cfg are:

- In setup.cfg, the – prefix must be excluded
- In setup.cfg, command line flags that take no arguments must be given an argument flag (1, T or TRUE for active, 0, F or FALSE for inactive)

Here's an example [nosetests] setup.cfg section:

```
[nosetests]
verbosity=1
detailed-errors=1
with-coverage=1
cover-package=nose
debug=nose.loader
pdb=1
pdb-failures=1
```

If you commonly run nosetests with a large number of options, using the nosetests setuptools command and configuring with setup.cfg can make running your tests much less tedious. (Note that the same options and format supported in setup.cfg are supported in all other config files, and the nosetests script will also load config files.)

Another reason to run tests with the command is that the command will install packages listed in your *tests_require*, as well as doing a complete build of your package before running tests. For packages with dependencies or that build C extensions, using the setuptools command can be more convenient than building by hand and running the nosetests script.

Bootstrapping

If you are distributing your project and want users to be able to run tests without having to install nose themselves, add nose to the setup_requires section of your setup():

```
setup(
    # ...
    setup_requires=['nose>=1.0']
    )
```

This will direct setuptools to download and activate nose during the setup process, making the `nosetests` command available.

`nose.commands.`**`get_user_options`**(*parser*)
convert a optparse option list into a distutils option tuple list

Twisted integration

This module provides a very simple way to integrate your tests with the Twisted event loop.

You must import this module *before* importing anything from Twisted itself!

Example:

```
from nose.twistedtools import reactor, deferred

@deferred()
def test_resolve():
    return reactor.resolve("www.python.org")
```

Or, more realistically:

```
@deferred(timeout=5.0)
def test_resolve():
    d = reactor.resolve("www.python.org")
    def check_ip(ip):
        assert ip == "67.15.36.43"
```

```
    d.addCallback(check_ip)
    return d
```

nose.twistedtools.**threaded_reactor**()
Start the Twisted reactor in a separate thread, if not already done. Returns the reactor. The thread will automatically be destroyed when all the tests are done.

nose.twistedtools.**deferred**(*timeout=None*)
By wrapping a test function with this decorator, you can return a twisted Deferred and the test will wait for the deferred to be triggered. The whole test function will run inside the Twisted event loop.

The optional timeout parameter specifies the maximum duration of the test. The difference with timed() is that timed() will still wait for the test to end, while deferred() will stop the test when its timeout has expired. The latter is more desireable when dealing with network tests, because the result may actually never arrive.

If the callback is triggered, the test has passed. If the errback is triggered or the timeout expires, the test has failed.

Example:

```
@deferred(timeout=5.0)
def test_resolve():
    return reactor.resolve("www.python.org")
```

Attention! If you combine this decorator with other decorators (like "raises"), deferred() must be called *first*!

In other words, this is good:

```
@raises(DNSLookupError)
@deferred()
def test_error():
    return reactor.resolve("xxxjhjhj.biz")
```

and this is bad:

```
@deferred()
@raises(DNSLookupError)
def test_error():
    return reactor.resolve("xxxjhjhj.biz")
```

nose.twistedtools.**stop_reactor**()
Stop the reactor and join the reactor thread until it stops. Call this function in teardown at the module or package level to reset the twisted system after your tests. You *must* do this if you mix tests using these tools and tests using twisted.trial.

Traceback inspector

Simple traceback introspection. Used to add additional information to AssertionErrors in tests, so that failure messages may be more informative.

class nose.inspector.**Expander**(*locals*, *globals*)
Simple expression expander. Uses tokenize to find the names and expands any that can be looked up in the frame.

nose.inspector.**find_inspectable_lines**(*lines*, *pos*)
Find lines in home that are inspectable.

Walk back from the err line up to 3 lines, but don't walk back over changes in indent level.

Walk forward up to 3 lines, counting separated lines as 1. Don't walk over changes in indent level (unless part of an extended line)

nose.inspector.**inspect_traceback**(*tb*)
Inspect a traceback and its frame, returning source for the expression where the exception was raised, with simple variable replacement performed and the line on which the exception was raised marked with '>>'

nose.inspector.**tbsource**(*tb*, *context=6*)
Get source from a traceback object.

A tuple of two things is returned: a list of lines of context from the source code, and the index of the current line within that list. The optional second argument specifies the number of lines of context to return, which are centered around the current line.

Note: This is adapted from inspect.py in the python 2.4 standard library, since a bug in the 2.3 version of inspect prevents it from correctly locating source lines in a traceback frame.

Utility functions

Utility functions and classes used by nose internally.

nose.util.**absdir**(*path*)
Return absolute, normalized path to directory, if it exists; None otherwise.

nose.util.**absfile**(*path*, *where=None*)
Return absolute, normalized path to file (optionally in directory where), or None if the file can't be found either in where or the current working directory.

nose.util.**file_like**(*name*)
A name is file-like if it is a path that exists, or it has a directory part, or it ends in .py, or it isn't a legal python identifier.

nose.util.**func_lineno**(*func*)
Get the line number of a function. First looks for compat_co_firstlineno, then func_code.co_first_lineno.

nose.util.**getfilename**(*package*, *relativeTo=None*)
Find the python source file for a package, relative to a particular directory (defaults to current working directory if not given).

nose.util.**getpackage**(*filename*)
Find the full dotted package name for a given python source file name. Returns None if the file is not a python source file.

```
>>> getpackage('foo.py')
'foo'
>>> getpackage('biff/baf.py')
'baf'
>>> getpackage('nose/util.py')
'nose.util'
```

Works for directories too.

```
>>> getpackage('nose')
'nose'
>>> getpackage('nose/plugins')
'nose.plugins'
```

And __init__ files stuck onto directories

```
>>> getpackage('nose/plugins/__init__.py')
'nose.plugins'
```

Absolute paths also work.

```
>>> path = os.path.abspath(os.path.join('nose', 'plugins'))
>>> getpackage(path)
'nose.plugins'
```

nose.util.**isclass**(*obj*)

Is obj a class? Inspect's isclass is too liberal and returns True for objects that can't be subclasses of anything.

nose.util.**ispackage**(*path*)

Is this path a package directory?

```
>>> ispackage('nose')
True
>>> ispackage('unit_tests')
False
>>> ispackage('nose/plugins')
True
>>> ispackage('nose/loader.py')
False
```

nose.util.**isproperty**(*obj*)

Is this a property?

```
>>> class Foo:
...     def got(self):
...         return 2
...     def get(self):
...         return 1
...     get = property(get)
```

```
>>> isproperty(Foo.got)
False
>>> isproperty(Foo.get)
True
```

nose.util.**ln**(*label*)

Draw a 70-char-wide divider, with label in the middle.

```
>>> ln('hello there')
'---------------------------- hello there -----------------------------'
```

class nose.util.**odict**(**arg*, ***kw*)

Simple ordered dict implementation, based on:

http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/107747

nose.util.**regex_last_key**(*regex*)

Sort key function factory that puts items that match a regular expression last.

```
>>> from nose.config import Config
>>> from nose.pyversion import sort_list
>>> c = Config()
>>> regex = c.testMatch
```

```
>>> entries = ['.', '..', 'a_test', 'src', 'lib', 'test', 'foo.py']
>>> sort_list(entries, regex_last_key(regex))
>>> entries
['.', '..', 'foo.py', 'lib', 'src', 'a_test', 'test']
```

nose.util.**resolve_name**(*name*, *module=None*)
Resolve a dotted name to a module and its parts. This is stolen wholesale from unittest.TestLoader.loadTestByName.

```
>>> resolve_name('nose.util')
<module 'nose.util' from...>
>>> resolve_name('nose.util.resolve_name')
<function resolve_name at...>
```

nose.util.**split_test_name**(*test*)
Split a test name into a 3-tuple containing file, module, and callable names, any of which (but not all) may be blank.

Test names are in the form:

file_or_module:callable

Either side of the : may be dotted. To change the splitting behavior, you can alter nose.util.split_test_re.

nose.util.**src**(*filename*)
Find the python source file for a .pyc, .pyo or $py.class file on jython. Returns the filename provided if it is not a python source file.

nose.util.**test_address**(*test*)
Find the test address for a test, which may be a module, filename, class, method or function.

nose.util.**tolist**(*val*)
Convert a value that may be a list or a (possibly comma-separated) string into a list. The exception: None is returned as None, not [None].

```
>>> tolist(["one", "two"])
['one', 'two']
>>> tolist("hello")
['hello']
>>> tolist("separate,values, with, commas,  spaces , are    ,ok")
['separate', 'values', 'with', 'commas', 'spaces', 'are', 'ok']
```

nose.util.**transplant_class**(*cls*, *module*)
Make a class appear to reside in *module*, rather than the module in which it is actually defined.

```
>>> from nose.failure import Failure
>>> Failure.__module__
'nose.failure'
>>> Nf = transplant_class(Failure, __name__)
>>> Nf.__module__
'nose.util'
>>> Nf.__name__
'Failure'
```

nose.util.**transplant_func**(*func*, *module*)
Make a function imported from module A appear as if it is located in module B.

```
>>> from pprint import pprint
>>> pprint.__module__
```

```
'pprint'
>>> pp = transplant_func(pprint, __name__)
>>> pp.__module__
'nose.util'
```

The original function is not modified.

```
>>> pprint.__module__
'pprint'
```

Calling the transplanted function calls the original.

```
>>> pp([1, 2])
[1, 2]
>>> pprint([1,2])
[1, 2]
```

nose.util.**try_run**(*obj*, *names*)

Given a list of possible method names, try to run them with the provided object. Keep going until something works. Used to run setup/teardown methods for module, package, and function tests.

Contributing to nose

You'd like to contribute to nose? Great! Now that nose is hosted under GitHub, contributing is even easier.

Get the code!

Start by getting a local working copy of nose from github:

```
git clone https://github.com/nose-devs/nose
```

If you plan to submit changes back to the core repository, you should set up a public fork of your own somewhere (GitHub is a good place to do that). See GitHub's help for details on how to contribute to a Git hosted project like nose.

Running nose's tests

nose runs its own test suite with *tox <http://codespeak.net/tox/>*. You don't have to install tox to run nose's test suite, but you should, because tox makes it easy to run all tests on all supported python versions. You'll also need python 2.4, 2.5, 2.6, 2.7, 3.1 and jython installed somewhere in your $PATH.

Discuss

Join the nose developer list at google groups. It's low-traffic and mostly signal.

What to work on?

You can find a list of open issues at nose's issue tracker. If you'd like to work on an issue, leave a comment on the issue detailing how you plan to fix it, or simply submit a pull request.

I have a great idea for a plugin...

Great! *Write it*. Release it on pypi. If it gains a large following, and becomes stable enough to work with nose's 6+ month release cycles, it may be a good candidate for inclusion in nose's builtin plugins.

What's new

1.3.8 (unreleased)

- Add option to suppress printing of coverage report Patch by Eric Larson.

1.3.7

- Fix loading packages from capitalised package on Windows Patch by Thomas Kluyver

1.3.6

- Re-release of 1.3.5 with wheels fixed.

1.3.5

- Fix #875: nose doesn't collect tests when subpackage is given as arg
- Fix #809: tests not discovered for namespace packages on Windows
- Fix #815: "ValueError: unsupported pickle protocol" with –with-id
- Wrap the working dir path name in quotes when reporting an error. Patch by Őry Máté.
- Fix #887: Fix a discrepancy in test names between Python 2 and Python 3
- Fix #131: Use os.stat() to check if file is executable Patch by Arnon Yaari.
- Fix #820 and #719: Fix coverage plugin with multiprocess Patch by Nicolas Grasset.

1.3.4

- Recognize doctest options defined in other plugins Patch by Daniel Lepage
- Another fix for Python 3.4: Call super in LazySuite to access _removed_tests variable Patch by Robert Kuska
- Fix for try_run when using bound methods Patch by Christian Lyder Jacobsen

1.3.3

- Fixed a minor issue with the reported version number.

1.3.2

- Fixed an issue where build_ext was not working under setup.py nosetest Patch by Michael Crusoe
- Fixed #786: generator method fails with callable instance Patch by Antoine Dechaume
- Fixed a traceback when using string exceptions
- Fixed #792: "Not a directory" error when using python setup.py nosetests
- Fixed #779: xunit report file is written in –where directory
- Fixed #782: Test failures with Python >= 3.3 Patch by Dmitry Shachnev
- Fixed #780: Fix a regression with Python 3
- Fixed #783: try_run is broken with Python 3.4

1.3.1

- The log capture plugin now correctly applies filters that were added using *addFilter*. Patch by Malthe Borch.
- Corrected a reference to the multiprocessing plugin in the documentation. Patch by Nick Loadholtes.
- Fixed #447: doctests fail when getpackage() returns None Patch by Matthew Brett.
- Fixed #749: xunit exceeds recursion limit Patch by André Caron.
- Fixed a number of unicode-related issues. Patch by John Szakmeister.
- Added the ability to ignore config files via an environment variable Patch by Lukasz Balcerzak
- Fixed #720: nose with detailed errors raises encoding error Patch by John Szakmeister. Thanks to Guillaume Ayoub for the test case.
- Fixed #692: UnicodeDecodeError in xunit when capturing stdout and stderr Patch by John Szakmeister.
- Fixed #693: Python 2.4 incompatibilities Patch by John Szakmeister.
- Don't save zero-byte xunit test reports Patch by Dan Savilonis.
- Fix Importer.importFromPath to be able to import modules whose names start with __init__ Patch by Paul Bonser.
- Add a fake isatty() method to Tee Patch by Jimmy Wennlund.
- Fix #700: Tee is missing the writelines() method Patch by John Szakmeister.
- Fix #649: UnicodeDecodeError when an exception contains encoded strings Patch by John Szakmeister.
- Fix #687: verbosity is not a flag Patch by John Szakmeister.
- Fixed a suppressed deprecation warning Patch by Arnon Yaari.
- Fixed some broken links in the documentation Patch by Arnon Yaari.
- Add missing format parameter in error message Patch by Etienne Millon.
- Switched docs to point at the GitHub site for the issue tracker Patch by Daniel Beck.
- Fix #447: doctests fail when getpackage() returns None Patch by Matthew Brett.
- Fix #366: make –pdb report on errors and failures. Use –pdb-error to get the old behavior. Patch by Arnon Yaari.
- Fix #501: Imported test generators are misrecognized as simple test functions Patch by John Szakmeister.
- Added a test for issue #501 Patch by Michael Killough.
- Use SkipTest from unittest2, if available, for better integration with testtools Patch by Ian Wienand.
- Fix #759: Test failures with Python 3.4 Patch by Barry Warsaw.
- Add a note about executable files in the usage, and how to workaround it Patch by Michael Dunn.
- Fix #743: fix an incorrect regex in writing_tests.rst Patch by Anne Moroney.
- Fix #690: Don't traceback when coverage reports fail. Patch by John Szakmeister.
- Display the report summary and stack traces even if Ctrl-C was pressed during the test run. Patch by Kevin Qiu.
- Fix #771: attr plugin is broken when parent and child share same method name with different attributes Patch by John Szakmeister. Test case provided by Thomas Grainger.
- Fix #728: attrib plugin rejects any staticmethod Patch by John Szakmesiter. Test case provided by Thomas Kluyver.
- Fix the plugin testing example. Patch by Charlie Dominio.
- Instruct coverage instance to cover requested packages only.

1.3.0

- Fix #556: fix selecting specific tests in the setuptools command. Patch by Andrey Golovizin.
- Fixed timed decorator to return the result of the wrapped function. Patch by Praful Mathur.
- Fixed #513: exception in test generator leads to a TypeError. Patch by Dmitry Shachnev.
- Fixed #535: nose.importer causes bizarre import errors if sys.path includes symlinks. Patch by Antoine Pitrou.
- Added support for Python 3.3. Patch by John Szakmeister and Bradley Froehle.
- Added the ability to skip generating .pyc files with –no-byte-complie. Patch by John Szakmeister.
- Suppress tracebacks caused by configuration errors (#401). Patch by Andrea Corbellini.
- Fixed doctest failures under Python 3 (#519). Patch by John Szakmeister.
- Fixed test_address's checking for builtins to work with Python 2 and 3. Patch by Joe Mathes and John Szakmeister.
- Fixed a bug where nose.tools was not being installed on systems without setuptools. (#554) Patch by Bradley Froehle.
- Fixed a bug in xunit that was causing it to die ungracefully under Python 3. (#134) Patch by John Szakmeister.
- Fixed #561: logcapture shouldn't buffer records, breaks %r point-in-time formatting. Patch by Santeri Paavolainen.
- Taught xunit to capture stdout and stderr in the xunit report. Patch by John Szakmeister.
- Repopulate nose.tools.__all__ so that the autodocs generate correctly. Patch by Taavi Burns.
- Fixed a bug where nose was failing to parse the NOSE_COVER_PACKAGE environment variable correctly. (#109) Patch by Churkin Oleg.
- Taught nose to lookup it's usage text according to PEP-302. Patch by Bradely Froehle.
- Fixed an ugly traceback when a test class was imported from another module, but was missing a method. (#595) Patch by Michael Williamson.
- Fix formatFailure expection if missing tb in exc_info. (#603) Patch by Sascha Peilicke.
- Taught the test suite to skip coverage tests if the coverage module is not available. (#597) Patch by Dmitry Shachnev.
- Fix #135: ResultProxy._prepareErr mangles error output in some cases. Patch by Arnon Yaari.
- Updated plugin documentation to fix numerous typos and incorrect names. Patch by Arnon Yaari.
- Added the ability to run nose as a module. Patch by Stefano Rivera.
- Fix installing Nose under Windows with Python 3.3. Patch by John Szakmeister.
- Added documentation for –processes=0 and the default value of –process-timeout. Patch by Takafumi Arakaki.
- Fixed broken references to non-existent documentation. Patch by Dmitry Shachnev.
- Fix –cover-min-percentage with –cover-branches. (#626) Patch by B. W. Baugh.
- Fix –cover-min-percentage with single packages. Patch by John Szakmeister.
- Fixed reference to PEP 8 to use Sphinx construct for better linking. Patch by Mahhtijs van der Vleuten.
- Fixed a reference to –cover-packages to use the correct option name. Patch by Wesley Baugh.
- Updated Nose to avoid using the deprecated compiler module when possible. Patch by Kim Scheilbel.
- Updated docstrings of formatFailure() and formatError(). Patch by Daniel Abel.

- Several man page fixes for better formatting. Patches by Dmitry Shachnev.
- Fixed a bug causing Nose to crash in directories that end in '.py'. (#612) Patch by Arnon Yaari.
- Fixed a traceback when a test raises SystemExit and has captured output. (#526) Patch by Arnon Yaari.
- Fixed ImportError running nosetests on namespace package in a subdirectory on Windows. (#19) Patch by Arnon Yaari.

1.2.1

- Correct nose.__version__ (#549). Thanks to Chris Withers for the bug report.

1.2.0

- Fixed issue where plugins included with *addplugins* keyword could be overridden by built-in plugins (or third-party plugins registered with setuptools) of the same name (#466). Patch by Brendan McCollam
- Adds `--cover-xml` and `--cover-xml-file` (#311). Patch by Timothée Peignier.
- Adds support for `--cover-branches` (related to #370). Patch by Timothée Peignier.
- Fixed Unicode issue on Python 3.1 with coverage (#442)
- fixed class level fixture handling in multiprocessing plugin
- Clue in the `unittest` module so it no longer prints traceback frames for our clones of their simple assertion helpers (#453). Patch by Erik Rose.
- Stop using the `assert` statement in `ok_` and `eq_` so they work under `python -O` (#504). Patch by Erik Rose.
- Add loglevel option to logcapture plugin (#493). Patch by Arach Tchoupani.
- Add doctest options flag (#7 from google code tracker). Patch by Michael Forbes.
- Add support for using 2to3 with the nosetests setuptools command. Patch by Andrey Golovizin.
- Add –cover-min-percentage flag to force test runs without sufficient coverage to fail (#540). Patch by Domen Kožar.
- Add travis-ci configuraion (#545). Patch by Domen Kožar.
- Call reactor.stop from twisted thread (#301). Patch by Adi Roiban.

1.1.2

- Fixed regression where the .coverage file was not saved (#439). Patch by Timothée Peignier.

1.1.1

- Fixed missing nose.sphinx module in source distribution (#436).

1.1.0

- Revised multiprocessing implementation so that it works for test generators (#399). Thanks to Rosen Diankov for the patch.
- More fixes to multiprocessing implemented by Buck Golemon and Gary Donovan (also part of #399).
- Lots of improvements to the attrib plugin by Bobby Impollonia (#412, #411, #324 and #381)
- Code coverage plugin now uses native HTML generation when coverage 3 is installed (#264). Thanks to Timothée Peignier for the patch.
- Xunit plugin now shows test run time in fractions of a second (#317)
- @attr (from nose.plugins.attrib) can now be used as a class decorator (#292)

- Fixes Xunit plugin to handle non-UTF8 characters (#395)
- Fixes Xunit plugin for reporting generator tests (#369)
- Fixed problems with SkipTest in Python 3.2 (#389)
- Fixed bug in doctest plugin under python 3. Thanks to Thomas Kluyver for the patch. (#391)
- Fixes mishandling of custom exceptions during failures (#405)
- Fixed subtle bug in `--first-package-wins` that made it unpredictable (#293)
- Fixes case where teardown_class() was called more than once (#408). Thanks to Heng Liu for the patch.
- Fixes coverage module names – 'cal' used to also match calendar which was a bug (#433)
- Fixes capture plugin when exception message contains non-ascii chars (#402)
- Fixed bug in tests for twisted tools. Thanks to Thomas Kluyver for the patch.
- Makes `--plugins` more succinct when there are no options (#235)

1.0.0

- Made nose compatible with python 3. **Huge** thanks to Alex "foogod" Stewart!

0.11.4

- Made nose compatible with Python 2.7.

0.11.3

- Fixed default plugin manager's use of plugin overriding. Thanks to rob.daylife for the bug report and patch. (#323).

0.11.2

- Changed plugin loading so that external plugins loaded via extension points can override builtin plugins with the same name.
- Updated multiprocess plugin and nose's packaging to allow multiprocess plugin to work on Windows (#265).
- Fixed bug in xunit plugin's interaction with suites and errors in module-level setup. Thanks to Mark McCague for the bug report (#279).
- Fixed bug in nose.loader.TestLoader that allowed Test classes that raise exceptions in __init__ to crash the test run (#269).
- Fixed bugs in nose's test suite that caused spurious failures on Windows.
- Fixed bug in twisted tools: delayed calls were not shut down on reactor stop. Thanks to abbeyj for the patch (#278).
- Fixed bug where root log handlers were not cleared. For example, this was emitting unwanted messages when testing Google App Engine websites.
- Fixed bug in test names output by xunit plugin. Thanks to Philip Jenvey for the bug report and patch (#280).
- Fixed bug in profile plugin that caused stats to fail to print under Python 2.5 and later. Thanks to djs at n-cube dot org for the bug report (#285).
- Improved logcapture filtering, with default setting to filter out log messages from nose itself. Thanks to gfxmonk for the patch (#277).
- The xunit plugin now tags skipped tests with a <skipped> testcase tag, and prevents the XML from containing invalid control characters.
- Updated nose to be compatible with python 2.7 (#305).

- Updated loading of usage document to allow nose to run from within an egg archive (#288).
- Fixed IronPython checks to make nose compatible with more versions of IronPython. Thanks to Kevin Mitchell for the patch (#316).

0.11.1

- Fixed bug in xunit plugin xml escaping. Thanks to Nat Williams for the bug report (#266).
- Fixed bug in xunit plugin that could cause test run to crash after certain types of errors or actions by other plugins.
- Fixed bug in testid plugin that could cause test run to crash after certain types of errors or actions by other plugins.
- Fixed bug in collect only plugin that caused it to fail when collecting from test generators.
- Fixed some broken html in docs.

0.11

- **All new documentation!** nose's documentation is now generated by Sphinx. And thanks to Pam Zerbinos, it is much better organized and easier to use and read.
- Two new core commandline options can help with testing namespace packages. `--first-package-wins` is useful when you want to test one part of a namespace package that uses another part; in previous versions of nose, the other part of the namespace package would be evicted from sys.modules when the 2nd loaded. `--traverse-namespace` is useful if you want nose to discover tests across entries in a package's __path__. (This was formerly the default behavior).
- To make it easier to use custom plugins without needing setuptools, `nose.core.main()` and `nose.core.run()` now support an *addplugins* keyword argument that takes a list of additional plugins to make available. **Note** that adding a plugin to this list **does not** activate or enable the plugin, only makes it available to be enabled via command-line or config file settings.
- Thanks to Kevin Mitchell, nose is now more compatible with IronPython. IronPython is still not fully supported, but it should work. If you'd like to improve nose's support for IronPython further, please join the nose developer's list and volunteer to become the IronPython maintainer for nose!
- Added multiprocess plugin that allows tests to be run in parallel across multiple processes.
- Added logcapture plugin that captures logging messages and prints them with failing tests. Thanks to Max Ischenko for the implementation.
- Added optional HTML coverage reports to coverage plugin. Thanks to Augie Fackler for the patch.
- Added plugin that enables collection of tests in all modules. Thanks to Peter Fein for the patch (#137).
- Added –failed option to testid plugin. When this option is in effect, if any tests failed in the previous test run (so long as testid was active for that test run) only the failed tests will run.
- Made it possible to 'yield test' in addition to 'yield test,' from test generators. Thanks to Chad Whitacre for the patch (#230).
- Fixed bug that caused traceback inspector to fail when source code file could not be found. Thanks to Philip Jenvey for the bug report and patch (#236).
- Fixed some issues limiting compatibility with IronPython. Thanks to Kevin Mitchell for the patch.
- Added support for module and test case fixtures in doctest files (#60).
- Added –traverse-namespace commandline option that restores old default behavior of following all package __path__ entries when loading tests from packages. Thanks to Philip Jenvey for the patch (#167).

- Added –first-package-wins commandline option to better support testing parts of namespace packages. Thanks to Jason Coombs for the bug report (#197).
- Added versioned nosetests scripts (#123).
- Fixed bug that would cause context teardown to fail to run in some cases. Thanks to John Shaw for the bug report and patch (#234).
- Enabled doctest plugin to use variable other than "_" as the default result variable. Thanks to Matt Good for the patch (#163).
- Fixed bug that would cause unicode output to crash output capture. Thanks to schickb for the bug report (#227).
- Added setUp and tearDown as valid names for module-level fixtures. Thanks to AgilityNerd for the patch (#211).
- Fixed bug in list of valid names for package-level fixtures. Thanks to Philip Jenvey for the patch (#237).
- Updated man page generation using hacked up manpage writer from docutils sandbox. Thanks grubert@users.sourceforge.net for the original module.

0.10.4

- nose is now compatible with python 2.6.

0.10.3

- Fixed bug in nosetests setup command that caused an exception to be raised if run with options. Thanks to Philip Jenvey for the bug report (#191).
- Raised score of coverage plugin to 200, so that it will execute before default-score plugins, and so be able to catch more import-time code. Thanks to Ned Batchelder for the bug report and patch (#190).

0.10.2

- nose now runs under jython (jython svn trunk only at this time). Thanks to Philip Jenvey, Pam Zerbinos and the other pycon sprinters (#160).
- Fixed bugs in loader, default plugin manager, and other modules that caused plugin exceptions to be swallowed (#152, #155). Thanks to John J Lee for the bug report and patch.
- Added selftest.py script, used to test a non-installed distribution of nose (#49). Thanks to Antoine Pitrou and John J Lee for the bug report and patches.
- Fixed bug in nose.importer that caused errors with namespace packages. Thanks to Philip Jenvey for the bug report and patch (#164).
- Fixed bug in nose.tools.with_setup that prevented use of multiple @with_setup decorators. Thanks to tlesher for the bug report (#151).
- Fixed bugs in handling of context fixtures for tests imported into a package. Thanks to Gary Bernhardt for the bug report (#145).
- Fixed bugs in handling of config files and config file options for plugins excluded by a RestrictedPluginManager. Thanks to John J Lee and Philip Jenvey for the bug reports and patches (#158, #166).
- Updated ErrorClass exception reporting to be shorter and more clear. Thanks to John J Lee for the patch (#142).
- Allowed plugins to load tests from modules starting with '_'. Thanks to John J Lee for the patch (#82).
- Updated documentation about building as rpm (#127).
- Updated config to make including executable files the default on IronPython as well as on Windows. Thanks to sanxiyn for the bug report and patch (#183).

- Fixed a python 2.3 incompatibility in errorclass_failure.rst (#173). Thanks to Philip Jenvey for the bug report and patch.
- Classes with metaclasses can now be collected as tests (#153).
- Made sure the document tree in the selector plugin test is accurate and tested (#144). Thanks to John J Lee for the bug report and patch.
- Fixed stack level used when dropping into pdb in a doctest (#154). Thanks to John J Lee for the bug report and patch.
- Fixed bug in ErrorClassPlugin that made some missing keyword argument errors obscure (#159). Thanks to Philip Jenvey for the bug report and patch.

0.10.1

- Fixed bug in capture plugin that caused it to record captured output on the test in the wrong attribute (#113).
- Fixed bug in result proxy that caused tests to fail if they accessed certain result attibutes directly (#114). Thanks to Neilen Marais for the bug report.
- Fixed bug in capture plugin that caused other error formatters changes to be lost if no output was captured (#124). Thanks to someone at ilorentz.org for the bug report.
- Fixed several bugs in the nosetests setup command that made some options unusable and the command itself unusable when no options were set (#125, #126, #128). Thanks to Alain Poirier for the bug reports.
- Fixed bug in handling of string errors (#130). Thanks to schl... at uni-oldenburg.de for the bug report.
- Fixed bug in coverage plugin option handling that prevented –cover-package=mod1,mod2 from working (#117). Thanks to Allen Bierbaum for the patch.
- Fixed bug in profiler plugin that prevented output from being produced when output capture was enabled on python 2.5 (#129). Thanks to James Casbon for the patch.
- Fixed bug in adapting 0.9 plugins to 0.10 (#119 part one). Thanks to John J Lee for the bug report and tests.
- Fixed bug in handling of argv in config and plugin test utilities (#119 part two). Thanks to John J Lee for the bug report and patch.
- Fixed bug where Failure cases due to invalid test name specifications were passed to plugins makeTest (#120). Thanks to John J Lee for the bug report and patch.
- Fixed bugs in doc css that mangled display in small windows. Thanks to Ben Hoyt for the bug report and Michal Kwiatkowski for the fix.
- Made it possible to pass a list or comma-separated string as defaultTest to main(). Thanks to Allen Bierbaum for the suggestion and patch.
- Fixed a bug in nose.selector and nose.util.getpackage that caused directories with names that are not legal python identifiers to be collected as packages (#143). Thanks to John J Lee for the bug report.

0.10.0

- Fixed bug that broke plugins with names containing underscores or hyphens. Thanks to John J Lee for the bug report and patch (Issue #81).
- Fixed typo in nose.__all__. Thanks to John J Lee for the bug report.
- Fixed handling of test descriptions that are multiline docstrings. Thanks to James Casbon for the patch (Issue #50).
- Improved documentation of doctest plugin to make it clear that entities may have doctests, or themselves be tests, but not both. Thanks to John J Lee for the bug report and patch (Issue #84).
- Made __file__ available in non-python-module doctests.

- Fixed bug that made it impossible for plugins to exclude package directories from test discovery (Issue #89). Thanks to John J Lee for the bug report and patch.
- Fixed bug that swallowed TypeError and AttributeError exceptions raised in some plugin calls (Issue #95). Thanks to John J Lee for the bug report.
- Fixed inconsistencies in many interfaces and docs. Thanks to John J Lee for many bug reports.
- Fixed bugs in rpm generation (Issue #96). Thanks to Mike Verdone for the bug report and http://bugs.python.org/issue644744 for the fix.
- Fixed inconsisent use of os.environ in plugin testing utilities. Thanks to John J Lee for the bug report and patch (Issue #97).
- Fixed bug in test_address that prevented use of nose.case.Test in doctests (Issue #100). Thanks to John J Lee for the bug report.
- Fixed bug in error class plugin that caused string exceptions to be masked (#Issue 101). Thanks to depaula for the bug report.
- Fixed bugs in tests and the profiler plugin when running under Windows (Issue #103). Thanks to Sidnei Da Silva for the bug report.
- Fixed bugs in plugin calls that caused exception swallowing (Issue #107). Thanks to John L Lee for the bug report and patch.
- Added more plugin example doctests. Thanks to Kumar McMillan and John L Lee for patches and lots of help.
- Changed default location of id file for TestId plugin from ~/.noseids to .noseids in the configured working directory.

0.10.0b1

- Added support for a description attribute on tests in function and method test cases. Most useful for generators: set the description attribute on the yielded function.
- Fixed incorrect signature of addSuccess() method in IPluginInterface. Thanks to nosexunit for the bug report. (Issue #68).
- Fixed isclass() function in nose.util so that it will not raise an exception when examining instances that have no accessible __bases__ attribute. (Issue #65).
- Fixed passing of tests to result; the test itself and not the wrapper was being passed, resulting in test description plugin hooks not being called. (Issue #71).
- Fixed bugs in FailureDetail and Capture plugins, and plugin manager and proxy uses of chainable plugin calls. Thanks to Ian Bicking for the bug report (Issue #72).
- Improved documentation.

0.10.0a2

- Changed entry point name to nose.plugins.0.10 – dashes and other non-word characters besides . are not allowed in entry point names. (Issue #67)
- Fixed loading of plugins from that entry point.
- Fixed backwards-compatibility issue in nose.util (is_generator was renamed isgenerator). (Issue #64)
- Fixed bug in –logging-config option. Thanks to anton_kr... at yahoo com for the bug report. (Issue #62)
- Fixed bug in handling of –where argument: first –where was not passed to loader as workingDir. Thanks to nosexunit for the bug report. (Issue #63).

0.10.0a1

- Rewrote test loader to be more drop-in compatible with unittest.TestLoader and to support a more user-friendly command line.
- Rewrote test runner and result classes to reduce duplication of effort.
- Revised configuration system to be less convoluted.
- Added nose.case.TestCase as universal wrapper for all testcases. Plugins always see instances of this class.
- Added a management layer to the plugin system, allowing for easy use of different plugin loading schemes. The default manager loads builtin plugins, 0.10 plugins under the setuptools entrypoint nose.plugins.0-10 and provides limited support for legacy plugins loaded under the entrypoint nose.plugins.
- Added plugin hooks in all phases of discovery, running and description.
- Converted several formely core features to plugins: output capture, assert introspection, pdb, and skipped and deprecated test support.
- Added id plugin that allows for easier specification of tests on the command line.
- Added ErrorClassPlugin base class to allow for easier authoring of plugins that handle errors, like the builtin skipped and deprecated test plugins.
- Added support for loading doctests from non-module files for all supported python versions.
- Added score property to plugins to allow plugins to execute in a defined order (higher score execute first).
- Expanded nose's own test suite to include a variety of functional tests.
- Fixed many bugs.

0.9.3

- Added support for user configuration files. Thanks to Antoine Pitrou for the patch.
- Fixed bug that caused profiler plugin to leak 0-byte temp files. Thanks to Antoine Pitrou for the patch.
- Made usage of temp files in profiler plugin more sensible. Thanks to Michael Sclenker for the bug report.
- Fixed bug that stopped loading of twisted TestCase tests. Thanks to Kumar McMillan for the bug report.
- Corrected man page location. Thanks to luke macken for the bug report and patch.
- Added with_setup to nose.tools.__all__. Thanks to Allen Bierbaum for the bug report.
- Altered plugin loading so that builtin plugins can be loaded without setuptools. Thanks to Allen Bierbaum for the suggestion.
- Fixed a bug in the doctests plugin that caused an error when multiple exclude arguments were specified. Thanks to mbeachy for the bug report and patch.

0.9.2

- Added nosetests setuptools command. Now you can run python setup.py nosetests and have access to all nose features and plugins. Thanks to James Casbon for the patch.
- Added make_decorator function to nose.tools. Used to construct decorator functions that are well-behaved and preserve as much of the original function's metadata as possible. Thanks to Antoine Pitrou for the patch.
- Added nose.twistedtools, contributed by Antoine Pitrou. This module adds @deferred decorator that makes it simple to write deferred tests, with or without timeouts.
- Added monkeypatch to doctests that keeps doctest from stepping on coverage when the two plugins are used together. Thanks to David Avraamides for the bug report.
- Added isolation plugin. Use this plugin to automatically restore sys.modules after each test module or package. Thanks to Michal Kwiatkowski for the feature request.

- Fixed bug where -vvvv turned off verbose logging instead of making it even more verbose. Thanks to Ian Bicking for the bug report.
- Fixed bug where assert inspection would fail when the trailing """ of a docstring was one of the inspected lines. Thanks to cito at online dot de for the bug report.
- Updated attrib plugin to allow selection of test methods by attributes of the test case class. Thanks to Jason Hildebrand for the patch.
- Improved compatibility with python 2.2. Thanks to Chad Whitacre for the patch.
- Fixed bug in handling of options from setup.cfg. Thanks to Kumar McMillan for the patch.
- Fixed bug in generator methods, where a generator method using an inline funciton would result in an AttributeError. Thanks to Antoine Pitrou for the bug report.
- Updated coverage plugin to ignore lines tagged with #pragma: no cover, matching the behavior of coverage.py on the command line. Thanks to Bill Zingler for the bug report.
- Added a man page for nosetests. Thanks to Gustavo Noronha Silva for the request and providing an example.

0.9.1

- New function nose.runmodule() finds and runs tests only in a single module, which defaults to __main__ (like unittest.main() or doctest.runmodule()). Thanks Greg Wilson for the suggestion.
- Multiple -w (–where) arguments can now be used in one command line, to find and run tests in multiple locations. Thanks Titus Brown for the suggestion.
- Multiple –include and –exclude arguments are now accepted in one command line. Thanks Michal Kwiatkowski for the feature request.
- Coverage will now include modules not imported by any test when using the new –cover-inclusive switch. Thanks James Casbon for the patch.
- module:TestClass test selections now properly select all tests in the test class.
- startTest and stopTest are now called in plugins at the beginning and end of test suites, including test modules, as well as individual tests. Thanks Michal Kwiatkowski for the suggestion.
- Fix bug in test selection when run as `python setup.py test`: 'test' was passing through and being used as the test name selection. Thanks Kumar McMillan for the bug report.
- Fix bug in handling of -x/–stop option where the test run would stop on skipped or deprecated tests. Thanks Kumar McMillan for the bug report.
- Fix bug in loading tests from projects with layouts that place modules in /lib or /src dirs and tests in a parallel /tests dir.
- Fix bug in python version detection. Thanks Kevin Dangoor for the bug report and fix.
- Fix log message in selector that could raise IndexError. Thanks Kumar McMillan for the bug report and patch.
- Fix bug in handling doctest extension arguments specified in environment and on command line. Thanks Ian Bicking for the bug report.
- Fix bug in running fixtures (setup/teardown) that are not functions, and report a better error message when a fixture is not callable. Thanks Ian Bicking for the bug report.

0.9.0

- More unit tests and better test coverage. Numerous bugfixes deriving from same.
- Make –exe option do what it says, and turn it on by default on Windows. Add –noexe option so windows users can turn if off.Thanks richard at artsalliancemedia dot com for the bug reports.

- Handle a working directory that happens to be in the middle of a package more gracefully. Thanks Max Ischenko for the bug report and test case.
- Fix bugs in test name comparison when a test module is specified whose name overlaps that of a non-test module. Thanks Max Ischenko for the bug report and test case.
- Fix warning spam when a non-existent test file is requested on the command line. Thanks Max Ischenko for the bug report.

0.9.0b2

- Allow –debug to set any logger to DEBUG. Thanks to casbon at gmail dot com for the patch.
- Fix doctest help, which was missing notes about the environment variables that it accepts. Thanks to Kumar McMillan for the patch.
- Restore sys.stdout after run() in nose.core. Thanks to Titus Brown for the bug report.
- Correct handling of trailing comma in attrib plugin args. Thanks Titus Brown for the patch.

0.9.0b1

- Fix bug in handling of OR conditions in attrib plugin. Thanks to Titus Brown for the bug report.
- Fix bug in nose.importer that would cause an attribute error when a local module shadowed a builtin, or other object in sys.modules, without a __file__ attribute. Thanks to casbon at gmail dot com for the bug report.
- Fix bug in nose.tools decorators that would cause decorated tests to appear with incorrect names in result output.

0.9.0a2

- In TestLoader, use inspect's isfunction() and ismethod() to filter functions and methods, instead of callable(). Thanks to Kumar McMillan for reporting the bug.
- Fix doctest plugin: return an empty iterable when no tests are found in a directory instead of None. Thanks to Kumar McMillan for the bug report and patch.
- Ignore executable python modules, unless run with –exe file. This is a partial defense against nose causing trouble by loading python modules that are not import-safe. The full defense: don't write modules that aren't import safe!
- Catch and warn about errors on plugin load instead of dying.
- Renamed builtin profile module from nose.plugins.profile to nose.plugins.prof to avoid shadowing stdlib profile.py module.

0.9.0a1

- Add support for plugins, with hooks for selecting, loading and reporting on tests. Doctest and coverage are now plugins.
- Add builtin plugins for profiling with hotshot, selecting tests by attribute (contributed by Mika Eloranta), and warning of missed tests specified on command line.
- Change command line test selection syntax to match unittest. Thanks to Titus Brown for the suggestion.
- Option to drop into pdb on error or failure.
- Option to stop running on first error or failure. Thanks to Kevin Dangoor for the suggestion.
- Support for doctests in files other than python modules (python 2.4 only)
- Reimplement base test selection as single self-contained class.
- Reimplement test loading as unittest-compatible TestLoader class.
- Remove all monkeypatching.

- Reimplement output capture and assert introspection support in unittest-compatible Result class.
- Better support for multiline constructions in assert introspection.
- More context output with assert introspections.
- Refactor setuptools test command support to use proxied result, which enables output capture and assert introspection support without monkeypatching. Thanks to Philip J. Eby for the suggestion and skeleton implementation.
- Add support for generators in test classes. Thanks to Jay Parlar for the suggestion and patch.
- Add nose.tools package with some helpful test-composition functions and decorators, including @raises, contributed by Scot Doyle.
- Reimplement nose.main (TestProgram) to have unittest-compatible signature.
- All-new import path handling. You can even turn it off! (If you don't, nose will ensure that all directories from which it imports anything are on sys.path before the import.)
- Logging package used for verbose logging.
- Support for skipped and deprecated tests.
- Configuration is no longer global.

0.8.7

- Add support for py.test-style test generators. Thanks to Jay Parlar for the suggestion.
- Fix bug in doctest discovery. Thanks to Richard Cooper for the bug report.
- Fix bug in output capture being appended to later exceptions. Thanks to Titus Brown for the patch that uncovered the bug.
- Fix bug(?) in Exception patch that caused masked hasattr/__getattr__ loops to either become actual infinite loops, or at least take so long to finally error out that they might as well be infinite.
- Add -m option to restrict test running to only tests in a particular package or module. Like the -f option, -m does not restrict test *loading*, only test *execution*.
- When loading and running a test module, ensure that the module's path is in sys.path for the duration of the run, not just while importing the module.
- Add id() method to all callable test classes, for greater unittest compatibility.

0.8.6

- Fix bug with coverage output when sys.modules contains entries without __file__ attributes
- Added -p (–cover-packages) switch that may be used to restrict coverage report to modules in the indicated package(s)

0.8.5

- Output capture and verbose assertion errors now work when run like 'python setup.py test', as advertised.
- Code coverage improvements: now coverage will be output for all modules imported by any means that were not in sys.modules at the start of the test run. By default, test modules will be excluded from the coverage report, but you can include them with the -t (–cover-tests) option.

0.8.4

- Fix bugs in handling of setup/teardown fixtures that could cause TypeError exceptions in fixtures to be silently ignored, or multiple fixtures of the same type to run. Thanks to Titus Brown for the bug report.

0.8.3

- Add -V (–version) switch to nosetests
- Fix bug where sys.path would not be set up correctly when running some tests, producing spurious import errors (Thanks to Titus Brown and Mike Thomson for the bug reports)
- For test classses not derived from unittest.TestCase, output (module.Class) "doc string" as test description, when method doc string is available (Thanks to David Keeney for the suggestion, even if this isn't quite what he meant)

0.8.2

- Revise import to bypass sys.path and manipulate sys.modules more intelligently, ensuring that the test module we think we are loading is the module we actually load, and that modules loaded by other imports are not reloaded without cause
- Allow test directories inside of packages. Formerly directories matching testMatch but lacking an __init__.py would cause an ImportError when located inside of packages
- Fix bugs in different handling of -f switch in combination with -w and -o

0.8.1

- Fix bug in main() that resulted in incorrect exit status for nosetests script when tests fail
- Add missing test files to MANIFEST.in
- Miscellaneous pylint cleanups

0.8

- Add doctest support
- Add optional code coverage support, using Ned Batchelder's coverage.py; activate with –coverage switch or NOSE_COVERAGE environment variable
- More informative error message on import error
- Fix bug where module setup could be called twice and teardown skipped for certain setup method names.
- main() returns success value, does not exit. run_exit() added to support old behavior; nosetests script now calls nose.run_exit()

0.7.5

- Fix bus error on exit
- Discover tests inside of non-TestCase classes that match testMatch
- Reorganize selftest: now selftest tests the output of a full nose run
- Add test_with_setup.py contributed by Kumar McMillan

0.7.2

- Refactor and correct bugs in discovery and test loading
- Reorganize and expand documentation
- Add -f (run this test file only) switch

0.7.1

- Bugfix release: test files in root of working directory were not being stripped of file extension before import.

0.7

- Change license to LGPL
- Major rework of output capture and assert introspection

- Improve test discovery: now finds tests in packages
- Replace -n switch ('no cwd') with -w switch ('look here')

0.6

- New nosetests script
- Allow specification of names on command line that are loadable but not directly loadable as modules (eg nosetests -o path/to/tests.py)
- Add optional py.test-like assert introspection. Thanks to Kevin Dangoor for the suggestion.
- Improvements to selftest

0.5.1

- Increased compatibility with python 2.3 (and maybe earlier)
- Increased compatibility with tests written for py.test: now calls module.setup_module(module) if module.setup_module() fails

Further reading

Using custom plugins without setuptools

If you have one or more custom plugins that you'd like to use with nose, but can't or don't want to register that plugin as a setuptools entrypoint, you can use the `addplugins` keyword argument to *`nose.core.main()`* or *`nose.core.run()`* to make the plugins available.

To do this you would construct a launcher script for nose, something like:

```
from nose import main
from yourpackage import YourPlugin, YourOtherPlugin

if __name__ == '__main__':
    nose.main(addplugins=[YourPlugin(), YourOtherPlugin()])
```

Here's an example. Say that you don't like the fact that the collect-only plugin outputs 'ok' for each test it finds; instead you want it to output 'maybe.' You could modify the plugin itself, or instead, create a Maybe plugin that transforms the output into your desired shape.

Without the plugin, we get 'ok.'

```
>>> import os
>>> support = os.path.join(os.path.dirname(__file__), 'support')
>>> from nose.plugins.plugintest import run_buffered as run
>>> argv = [__file__, '-v', support] # --collect-only
>>> run(argv=argv)
test.test ... ok

----------------------------------------------------------------------
Ran 1 test in ...s

OK
```

Without '-v', we get a dot.

```
>>> run(argv=[__file__, support])
.
----------------------------------------------------------------------
Ran 1 test in ...s

OK
```

The plugin is simple. It captures and wraps the test result output stream and replaces 'ok' with 'maybe' and '.' with '?'.

```
>>> from nose.plugins.base import Plugin
>>> class Maybe(Plugin):
...     def setOutputStream(self, stream):
...         self.stream = stream
...         return self
...     def flush(self):
...         self.stream.flush()
...     def writeln(self, out=""):
...         self.write(out + "\n")
...     def write(self, out):
...         if out == "ok\n":
...             out = "maybe\n"
...         elif out == ".":
...             out = "?"
...         self.stream.write(out)
```

To activate the plugin, we pass an instance in the addplugins list.

```
>>> run(argv=argv + ['--with-maybe'], addplugins=[Maybe()])
test.test ... maybe

----------------------------------------------------------------------
Ran 1 test in ...s

OK
```

```
>>> run(argv=[__file__, support, '--with-maybe'], addplugins=[Maybe()])
?
----------------------------------------------------------------------
Ran 1 test in ...s

OK
```

Doctest Fixtures

Doctest files, like other tests, can be made more efficient or meaningful or at least easier to write by judicious use of fixtures. nose supports limited fixtures for use with doctest files.

Module-level fixtures

Fixtures for a doctest file may define any or all of the following methods for module-level setup:

- setup
- setup_module

- setupModule
- setUpModule

Each module-level setup function may optionally take a single argument, the fixtures module itself.

Example:

```
def setup_module(module):
    module.called[:] = []
```

Similarly, module-level teardown methods are available, which also optionally take the fixtures module as an argument:

- teardown
- teardown_module
- teardownModule
- tearDownModule

Example:

```
def teardown_module(module):
    module.called[:] = []
    module.done = True
```

Module-level setup executes **before any tests are loaded** from the doctest file. This is the right place to raise `nose.plugins.skip.SkipTest`, for example.

Test-level fixtures

In addition to module-level fixtures, *test*-level fixtures are supported. Keep in mind that in the doctest lexicon, the *test* is the *entire doctest file* – not each individual example within the file. So, like the module-level fixtures, test-level fixtures execute *once per file*. The differences are that:

- test-level fixtures execute **after** tests have been loaded, but **before** any tests have executed.
- test-level fixtures receive the doctest `doctest.DocFileCase` loaded from the file as their one *required* argument.

setup_test(test) is called before the test is run.

Example:

```
def setup_test(test):
    called.append(test)
    test.globs['count'] = len(called)
setup_test.__test__ = False
```

teardown_test(test) is alled after the test, unless setup raised an uncaught exception. The argument is the `doctest.DocFileCase` object, *not* a unittest.TestCase.

Example:

```
def teardown_test(test):
    pass
teardown_test.__test__ = False
```

Bottom line: setup_test, teardown_test have access to the *doctest test*, while setup, setup_module, etc have access to the *fixture* module. setup_module runs before tests are loaded, setup_test after.

Note: As in the examples, it's a good idea to tag your setup_test/teardown_test functions with `__test__ = False` to avoid them being collected as tests.

Lastly, the fixtures for a doctest file may supply a **globs(globs)** function. The dict returned by this function will be passed to the doctest runner as the globals available to the test. You can use this, for example, to easily inject a module's globals into a doctest that has been moved from the module to a separate file.

Example

This doctest has some simple fixtures:

```
called = []

def globs(globs):
    globs['something'] = 'Something?'
    return globs

def setup_module(module):
    module.called[:] = []

def setup_test(test):
    called.append(test)
    test.globs['count'] = len(called)
setup_test.__test__ = False

def teardown_test(test):
    pass
teardown_test.__test__ = False
```

The `globs` defined in the fixtures make the variable `something` available in all examples.

```
>>> something
'Something?'
```

The `count` variable is injected by the test-level fixture.

```
>>> count
1
```

Warning: This whole file is one doctest test. setup_test doesn't do what you think! It exists to give you access to the test case and examples, but it runs *once*, before all of them, not before each.

```
>>> count
1
```

Thus, `count` stays 1 throughout the test, no matter how many examples it includes.

Running Initialization Code Before the Test Run

Many applications, especially those using web frameworks like Pylons or Django, can't be tested without first being configured or otherwise initialized. Plugins can fulfill this requirement by implementing *`begin()`*.

In this example, we'll use a very simple example: a widget class that can't be tested without a configuration.

Here's the widget class. It's configured at the class or instance level by setting the `cfg` attribute to a dictionary.

```
>>> class ConfigurableWidget(object):
...     cfg = None
...     def can_frobnicate(self):
...         return self.cfg.get('can_frobnicate', True)
...     def likes_cheese(self):
...         return self.cfg.get('likes_cheese', True)
```

The tests verify that the widget's methods can be called without raising any exceptions.

```
>>> import unittest
>>> class TestConfigurableWidget(unittest.TestCase):
...     longMessage = False
...     def setUp(self):
...         self.widget = ConfigurableWidget()
...     def test_can_frobnicate(self):
...         """Widgets can frobnicate (or not)"""
...         self.widget.can_frobnicate()
...     def test_likes_cheese(self):
...         """Widgets might like cheese"""
...         self.widget.likes_cheese()
...     def shortDescription(self): # 2.7 compat
...         try:
...             doc = self._testMethodDoc
...         except AttributeError:
...             # 2.4 compat
...             doc = self._TestCase__testMethodDoc
...         return doc and doc.split("\n")[0].strip() or None
```

The tests are bundled into a suite that we can pass to the test runner.

```
>>> def suite():
...     return unittest.TestSuite([
...         TestConfigurableWidget('test_can_frobnicate'),
...         TestConfigurableWidget('test_likes_cheese')])
```

When we run tests without first configuring the ConfigurableWidget, the tests fail.

Note: The function `nose.plugins.plugintest.run()` reformats test result output to remove timings, which will vary from run to run, and redirects the output to stdout.

```
>>> from nose.plugins.plugintest import run_buffered as run
```

```
>>> argv = [__file__, '-v']
>>> run(argv=argv, suite=suite())
Widgets can frobnicate (or not) ... ERROR
Widgets might like cheese ... ERROR

======================================================================
ERROR: Widgets can frobnicate (or not)
----------------------------------------------------------------------
Traceback (most recent call last):
...
AttributeError: 'NoneType' object has no attribute 'get'
```

```
======================================================================
ERROR: Widgets might like cheese
----------------------------------------------------------------------
Traceback (most recent call last):
...
AttributeError: 'NoneType' object has no attribute 'get'

----------------------------------------------------------------------
Ran 2 tests in ...s

FAILED (errors=2)
```

To configure the widget system before running tests, write a plugin that implements *begin()* and initializes the system with a hard-coded configuration. (Later, we'll write a better plugin that accepts a command-line argument specifying the configuration file.)

```
>>> from nose.plugins import Plugin
>>> class ConfiguringPlugin(Plugin):
...     enabled = True
...     def configure(self, options, conf):
...         pass # always on
...     def begin(self):
...         ConfigurableWidget.cfg = {}
```

Now configure and execute a new test run using the plugin, which will inject the hard-coded configuration.

```
>>> run(argv=argv, suite=suite(),
...     plugins=[ConfiguringPlugin()])
Widgets can frobnicate (or not) ... ok
Widgets might like cheese ... ok

----------------------------------------------------------------------
Ran 2 tests in ...s

OK
```

This time the tests pass, because the widget class is configured.

But the ConfiguringPlugin is pretty lame – the configuration it installs is hard coded. A better plugin would allow the user to specify a configuration file on the command line:

```
>>> class BetterConfiguringPlugin(Plugin):
...     def options(self, parser, env={}):
...         parser.add_option('--widget-config', action='store',
...                           dest='widget_config', default=None,
...                           help='Specify path to widget config file')
...     def configure(self, options, conf):
...         if options.widget_config:
...             self.load_config(options.widget_config)
...             self.enabled = True
...     def begin(self):
...         ConfigurableWidget.cfg = self.cfg
...     def load_config(self, path):
...         from ConfigParser import ConfigParser
...         p = ConfigParser()
...         p.read([path])
...         self.cfg = dict(p.items('DEFAULT'))
```

To use the plugin, we need a config file.

```
>>> import os
>>> cfg_path = os.path.join(os.path.dirname(__file__), 'example.cfg')
>>> cfg_file = open(cfg_path, 'w')
>>> bytes = cfg_file.write("""\
... [DEFAULT]
... can_frobnicate = 1
... likes_cheese = 0
... """)
>>> cfg_file.close()
```

Now we can execute a test run using that configuration, after first resetting the widget system to an unconfigured state.

```
>>> ConfigurableWidget.cfg = None
>>> argv = [__file__, '-v', '--widget-config', cfg_path]
>>> run(argv=argv, suite=suite(),
...     plugins=[BetterConfiguringPlugin()])
Widgets can frobnicate (or not) ... ok
Widgets might like cheese ... ok

----------------------------------------------------------------------
Ran 2 tests in ...s

OK
```

Excluding Unwanted Packages

Normally, nose discovery descends into all packages. Plugins can change this behavior by implementing `IPluginInterface.wantDirectory()`.

In this example, we have a wanted package called `wanted_package` and an unwanted package called `unwanted_package`.

```
>>> import os
>>> support = os.path.join(os.path.dirname(__file__), 'support')
>>> support_files = [d for d in os.listdir(support)
...                  if not d.startswith('.')]
>>> support_files.sort()
>>> support_files
['unwanted_package', 'wanted_package']
```

When we run nose normally, tests are loaded from both packages.

Note: The function `nose.plugins.plugintest.run()` reformats test result output to remove timings, which will vary from run to run, and redirects the output to stdout.

```
>>> from nose.plugins.plugintest import run_buffered as run
```

```
>>> argv = [__file__, '-v', support]
>>> run(argv=argv)
unwanted_package.test_spam.test_spam ... ok
wanted_package.test_eggs.test_eggs ... ok

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

```
Ran 2 tests in ...s

OK
```

To exclude the tests in the unwanted package, we can write a simple plugin that implements `IPluginInterface.wantDirectory()` and returns `False` if the basename of the directory is `"unwanted_package"`. This will prevent nose from descending into the unwanted package.

```
>>> from nose.plugins import Plugin
>>> class UnwantedPackagePlugin(Plugin):
...     # no command line arg needed to activate plugin
...     enabled = True
...     name = "unwanted-package"
...
...     def configure(self, options, conf):
...         pass # always on
...
...     def wantDirectory(self, dirname):
...         want = None
...         if os.path.basename(dirname) == "unwanted_package":
...             want = False
...         return want
```

In the next test run we use the plugin, and the unwanted package is not discovered.

```
>>> run(argv=argv,
...     plugins=[UnwantedPackagePlugin()])
wanted_package.test_eggs.test_eggs ... ok

----------------------------------------------------------------------
Ran 1 test in ...s

OK
```

nose.plugins.plugintest, os.environ and sys.argv

nose.plugins.plugintest.PluginTester and `nose.plugins.plugintest.run()` are utilities for testing nose plugins. When testing plugins, it should be possible to control the environment seen plugins under test, and that environment should never be affected by `os.environ` or `sys.argv`.

```
>>> import os
>>> import sys
>>> import unittest
>>> import nose.config
>>> from nose.plugins import Plugin
>>> from nose.plugins.builtin import FailureDetail, Capture
>>> from nose.plugins.plugintest import PluginTester
```

Our test plugin takes no command-line arguments and simply prints the environment it's given by nose.

```
>>> class PrintEnvPlugin(Plugin):
...     name = "print-env"
...
...     # no command line arg needed to activate plugin
...     enabled = True
...     def configure(self, options, conf):
```

```
...         if not self.can_configure:
...             return
...         self.conf = conf
...
...     def options(self, parser, env={}):
...         print "env:", env
```

To test the argv, we use a config class that prints the argv it's given by nose. We need to monkeypatch nose.config.Config, so that we can test the cases where that is used as the default.

```
>>> old_config = nose.config.Config
>>> class PrintArgvConfig(old_config):
...
...     def configure(self, argv=None, doc=None):
...         print "argv:", argv
...         old_config.configure(self, argv, doc)
>>> nose.config.Config = PrintArgvConfig
```

The class under test, PluginTester, is designed to be used by subclassing.

```
>>> class Tester(PluginTester):
...     activate = "-v"
...     plugins = [PrintEnvPlugin(),
...                FailureDetail(),
...                Capture(),
...                ]
...
...     def makeSuite(self):
...         return unittest.TestSuite(tests=[])
```

For the purposes of this test, we need a known `os.environ` and `sys.argv`.

```
>>> old_environ = os.environ
>>> old_argv = sys.argv
>>> os.environ = {"spam": "eggs"}
>>> sys.argv = ["spamtests"]
```

PluginTester always uses the [nosetests, self.activate] as its argv. If `env` is not overridden, the default is an empty `env`.

```
>>> tester = Tester()
>>> tester.setUp()
argv: ['nosetests', '-v']
env: {}
```

An empty `env` is respected...

```
>>> class EmptyEnvTester(Tester):
...     env = {}
>>> tester = EmptyEnvTester()
>>> tester.setUp()
argv: ['nosetests', '-v']
env: {}
```

... as is a non-empty `env`.

```
>>> class NonEmptyEnvTester(Tester):
...     env = {"foo": "bar"}
>>> tester = NonEmptyEnvTester()
>>> tester.setUp()
argv: ['nosetests', '-v']
env: {'foo': 'bar'}
```

`nose.plugins.plugintest.run()` should work analogously.

```
>>> from nose.plugins.plugintest import run_buffered as run
>>> run(suite=unittest.TestSuite(tests=[]),
...      plugins=[PrintEnvPlugin()])
argv: ['nosetests', '-v']
env: {}

----------------------------------------------------------------------
Ran 0 tests in ...s

OK
>>> run(env={},
...      suite=unittest.TestSuite(tests=[]),
...      plugins=[PrintEnvPlugin()])
argv: ['nosetests', '-v']
env: {}

----------------------------------------------------------------------
Ran 0 tests in ...s

OK
>>> run(env={"foo": "bar"},
...      suite=unittest.TestSuite(tests=[]),
...      plugins=[PrintEnvPlugin()])
argv: ['nosetests', '-v']
env: {'foo': 'bar'}

----------------------------------------------------------------------
Ran 0 tests in ...s

OK
```

An explicit argv parameter is honoured:

```
>>> run(argv=["spam"],
...      suite=unittest.TestSuite(tests=[]),
...      plugins=[PrintEnvPlugin()])
argv: ['spam']
env: {}

----------------------------------------------------------------------
Ran 0 tests in ...s

OK
```

An explicit config parameter with an env is honoured:

```
>>> from nose.plugins.manager import PluginManager
>>> manager = PluginManager(plugins=[PrintEnvPlugin()])
>>> config = PrintArgvConfig(env={"foo": "bar"}, plugins=manager)
```

```
>>> run(config=config,
...     suite=unittest.TestSuite(tests=[]))
argv: ['nosetests', '-v']
env: {'foo': 'bar'}

----------------------------------------------------------------------
Ran 0 tests in ...s

OK
```

Clean up.

```
>>> os.environ = old_environ
>>> sys.argv = old_argv
>>> nose.config.Config = old_config
```

When Plugins Fail

Plugin methods should not fail silently. When a plugin method raises an exception before or during the execution of a test, the exception will be wrapped in a *nose.failure.Failure* instance and appear as a failing test. Exceptions raised at other times, such as in the preparation phase with `prepareTestLoader` or `prepareTestResult`, or after a test executes, in `afterTest` will stop the entire test run.

```
>>> import os
>>> import sys
>>> from nose.plugins import Plugin
>>> from nose.plugins.plugintest import run_buffered as run
```

Our first test plugins take no command-line arguments and raises AttributeError in beforeTest and afterTest.

```
>>> class EnabledPlugin(Plugin):
...     """Plugin that takes no command-line arguments"""
...
...     enabled = True
...
...     def configure(self, options, conf):
...         pass
...     def options(self, parser, env={}):
...         pass
>>> class FailBeforePlugin(EnabledPlugin):
...     name = "fail-before"
...
...     def beforeTest(self, test):
...         raise AttributeError()
>>> class FailAfterPlugin(EnabledPlugin):
...     name = "fail-after"
...
...     def afterTest(self, test):
...         raise AttributeError()
```

Running tests with the fail-before plugin enabled will result in all tests failing.

```
>>> support = os.path.join(os.path.dirname(__file__), 'support')
>>> suitepath = os.path.join(support, 'test_spam.py')
>>> run(argv=['nosetests', suitepath],
...     plugins=[FailBeforePlugin()])
```

```
EE
======================================================================
ERROR: test_spam.test_spam
----------------------------------------------------------------------
Traceback (most recent call last):
...
AttributeError

======================================================================
ERROR: test_spam.test_eggs
----------------------------------------------------------------------
Traceback (most recent call last):
...
AttributeError

----------------------------------------------------------------------
Ran 0 tests in ...s

FAILED (errors=2)
```

But with the fail-after plugin, the entire test run will fail.

```
>>> run(argv=['nosetests', suitepath],
...      plugins=[FailAfterPlugin()])
Traceback (most recent call last):
...
AttributeError
```

Likewise, since the next plugin fails in a preparatory method, outside of test execution, the entire test run fails when the plugin is used.

```
>>> class FailPreparationPlugin(EnabledPlugin):
...     name = "fail-prepare"
...
...     def prepareTestLoader(self, loader):
...         raise TypeError("That loader is not my type")
>>> run(argv=['nosetests', suitepath],
...      plugins=[FailPreparationPlugin()])
Traceback (most recent call last):
...
TypeError: That loader is not my type
```

Even AttributeErrors and TypeErrors are not silently suppressed as they used to be for some generative plugin methods (issue152).

These methods caught TypeError and AttributeError and did not record the exception, before issue152 was fixed: .loadTestsFromDir(), .loadTestsFromModule(), .loadTestsFromTestCase(), loadTestsFromTestClass, and .makeTest(). Now, the exception is caught, but logged as a Failure.

```
>>> class FailLoadPlugin(EnabledPlugin):
...     name = "fail-load"
...
...     def loadTestsFromModule(self, module):
...         # we're testing exception handling behaviour during
...         # iteration, so be a generator function, without
...         # actually yielding any tests
...         if False:
...             yield None
```

```
...         raise TypeError("bug in plugin")
>>> run(argv=['nosetests', suitepath],
...      plugins=[FailLoadPlugin()])
..E
======================================================================
ERROR: Failure: TypeError (bug in plugin)
----------------------------------------------------------------------
Traceback (most recent call last):
...
TypeError: bug in plugin

----------------------------------------------------------------------
Ran 3 tests in ...s

FAILED (errors=1)
```

Also, before issue152 was resolved, .loadTestsFromFile() and .loadTestsFromName() didn't catch these errors at all, so the following test would crash nose:

```
>>> class FailLoadFromNamePlugin(EnabledPlugin):
...     name = "fail-load-from-name"
...
...     def loadTestsFromName(self, name, module=None, importPath=None):
...         if False:
...             yield None
...         raise TypeError("bug in plugin")
>>> run(argv=['nosetests', suitepath],
...      plugins=[FailLoadFromNamePlugin()])
E
======================================================================
ERROR: Failure: TypeError (bug in plugin)
----------------------------------------------------------------------
Traceback (most recent call last):
...
TypeError: bug in plugin

----------------------------------------------------------------------
Ran 1 test in ...s

FAILED (errors=1)
```

Minimal plugin

Plugins work as long as they implement the minimal interface required by nose.plugins.base. They do not have to derive from nose.plugins.Plugin.

```
>>> class NullPlugin(object):
...
...     enabled = True
...     name = "null"
...     score = 100
...
...     def options(self, parser, env):
...         pass
...
...     def configure(self, options, conf):
```

```
...         pass
>>> import unittest
>>> from nose.plugins.plugintest import run_buffered as run
>>> run(suite=unittest.TestSuite(tests=[]),
...     plugins=[NullPlugin()])
----------------------------------------------------------------------
Ran 0 tests in ...s

OK
```

Plugins can derive from nose.plugins.base and do nothing except set a name.

```
>>> import os
>>> from nose.plugins import Plugin
>>> class DerivedNullPlugin(Plugin):
...
...     name = "derived-null"
```

Enabled plugin that's otherwise empty

```
>>> class EnabledDerivedNullPlugin(Plugin):
...
...     enabled = True
...     name = "enabled-derived-null"
...
...     def options(self, parser, env=os.environ):
...         pass
...
...     def configure(self, options, conf):
...         if not self.can_configure:
...             return
...         self.conf = conf
>>> run(suite=unittest.TestSuite(tests=[]),
...     plugins=[DerivedNullPlugin(), EnabledDerivedNullPlugin()])
...
----------------------------------------------------------------------
Ran 0 tests in ...s

OK
```

Failure of Errorclasses

Errorclasses (skips, deprecations, etc.) define whether or not they represent test failures.

```
>>> import os
>>> import sys
>>> from nose.plugins.plugintest import run_buffered as run
>>> from nose.plugins.skip import Skip
>>> from nose.plugins.deprecated import Deprecated
>>> support = os.path.join(os.path.dirname(__file__), 'support')
>>> sys.path.insert(0, support)
>>> from errorclass_failure_plugin import Todo, TodoPlugin, \
...                                       NonFailureTodoPlugin
>>> todo_test = os.path.join(support, 'errorclass_failing_test.py')
>>> misc_test = os.path.join(support, 'errorclass_tests.py')
```

nose.plugins.errorclass.ErrorClass has an argument `isfailure`. With a true isfailure, when the errorclass' exception is raised by a test, tracebacks are printed.

```
>>> run(argv=["nosetests", "-v", "--with-todo", todo_test],
...      plugins=[TodoPlugin()])
errorclass_failing_test.test_todo ... TODO: fix me
errorclass_failing_test.test_2 ... ok

======================================================================
TODO: errorclass_failing_test.test_todo
----------------------------------------------------------------------
Traceback (most recent call last):
...
Todo: fix me

----------------------------------------------------------------------
Ran 2 tests in ...s

FAILED (TODO=1)
```

Also, `--stop` stops the test run.

```
>>> run(argv=["nosetests", "-v", "--with-todo", "--stop", todo_test],
...      plugins=[TodoPlugin()])
errorclass_failing_test.test_todo ... TODO: fix me

======================================================================
TODO: errorclass_failing_test.test_todo
----------------------------------------------------------------------
Traceback (most recent call last):
...
Todo: fix me

----------------------------------------------------------------------
Ran 1 test in ...s

FAILED (TODO=1)
```

With a false .isfailure, errorclass exceptions raised by tests are treated as "ignored errors." For ignored errors, tracebacks are not printed, and the test run does not stop.

```
>>> run(argv=["nosetests", "-v", "--with-non-failure-todo", "--stop",
...           todo_test],
...      plugins=[NonFailureTodoPlugin()])
errorclass_failing_test.test_todo ... TODO: fix me
errorclass_failing_test.test_2 ... ok

----------------------------------------------------------------------
Ran 2 tests in ...s

OK (TODO=1)
```

Exception detail strings of errorclass errors are always printed when -v is in effect, regardless of whether the error is ignored. Note that exception detail strings may have more than one line.

```
>>> run(argv=["nosetests", "-v", "--with-todo", misc_test],
...      plugins=[TodoPlugin(), Skip(), Deprecated()])
...
```

```
errorclass_tests.test_todo ... TODO: fix me
errorclass_tests.test_2 ... ok
errorclass_tests.test_3 ... SKIP: skipety-skip
errorclass_tests.test_4 ... SKIP
errorclass_tests.test_5 ... DEPRECATED: spam
eggs

spam
errorclass_tests.test_6 ... DEPRECATED: spam

======================================================================
TODO: errorclass_tests.test_todo
----------------------------------------------------------------------
Traceback (most recent call last):
...
Todo: fix me

----------------------------------------------------------------------
Ran 6 tests in ...s

FAILED (DEPRECATED=2, SKIP=2, TODO=1)
```

Without -v, the exception detail strings are only displayed if the error is not ignored (otherwise, there's no traceback).

```
>>> run(argv=["nosetests", "--with-todo", misc_test],
...     plugins=[TodoPlugin(), Skip(), Deprecated()])
...
T.SSDD
======================================================================
TODO: errorclass_tests.test_todo
----------------------------------------------------------------------
Traceback (most recent call last):
...
Todo: fix me

----------------------------------------------------------------------
Ran 6 tests in ...s

FAILED (DEPRECATED=2, SKIP=2, TODO=1)
```

```
>>> sys.path.remove(support)
```

Importing Tests

When a package imports tests from another package, the tests are **completely** relocated into the importing package. This means that the fixtures from the source package are **not** run when the tests in the importing package are executed.

For example, consider this collection of packages:

```
>>> import os
>>> support = os.path.join(os.path.dirname(__file__), 'support')
>>> from nose.util import ls_tree
>>> print ls_tree(support)
|-- package1
|   |-- __init__.py
|   `-- test_module.py
```

```
|-- package2c
|   |-- __init__.py
|   `-- test_module.py
`-- package2f
    |-- __init__.py
    `-- test_module.py
```

In these packages, the tests are all defined in package1, and are imported into package2f and package2c.

Note: The run() function in *nose.plugins.plugintest* reformats test result output to remove timings, which will vary from run to run, and redirects the output to stdout.

```
>>> from nose.plugins.plugintest import run_buffered as run
```

---

package1 has fixtures, which we can see by running all of the tests. Note below that the test names reflect the modules into which the tests are imported, not the source modules.

```
>>> argv = [__file__, '-v', support]
>>> run(argv=argv)
package1 setup
test (package1.test_module.TestCase) ... ok
package1.test_module.TestClass.test_class ... ok
package1.test_module.test_function ... ok
package2c setup
test (package2c.test_module.TestCase) ... ok
package2c.test_module.TestClass.test_class ... ok
package2f setup
package2f.test_module.test_function ... ok

----------------------------------------------------------------------
Ran 6 tests in ...s

OK
```

When tests are run in package2f or package2c, only the fixtures from those packages are executed.

```
>>> argv = [__file__, '-v', os.path.join(support, 'package2f')]
>>> run(argv=argv)
package2f setup
package2f.test_module.test_function ... ok

----------------------------------------------------------------------
Ran 1 test in ...s

OK
>>> argv = [__file__, '-v', os.path.join(support, 'package2c')]
>>> run(argv=argv)
package2c setup
test (package2c.test_module.TestCase) ... ok
package2c.test_module.TestClass.test_class ... ok

----------------------------------------------------------------------
Ran 2 tests in ...s

OK
```

This also applies when only the specific tests are selected via the command-line.

```
>>> argv = [__file__, '-v',
...         os.path.join(support, 'package2c', 'test_module.py') +
...         ':TestClass.test_class']
>>> run(argv=argv)
package2c setup
package2c.test_module.TestClass.test_class ... ok

----------------------------------------------------------------------
Ran 1 test in ...s

OK
>>> argv = [__file__, '-v',
...         os.path.join(support, 'package2c', 'test_module.py') +
...         ':TestCase.test']
>>> run(argv=argv)
package2c setup
test (package2c.test_module.TestCase) ... ok

----------------------------------------------------------------------
Ran 1 test in ...s

OK
>>> argv = [__file__, '-v',
...         os.path.join(support, 'package2f', 'test_module.py') +
...         ':test_function']
>>> run(argv=argv)
package2f setup
package2f.test_module.test_function ... ok

----------------------------------------------------------------------
Ran 1 test in ...s

OK
```

Parallel Testing with nose

Note: Use of the multiprocess plugin on python 2.5 or earlier requires the multiprocessing module, available from PyPI and at http://code.google.com/p/python-multiprocessing/.

Using the *nose.plugins.multiprocess* plugin, you can parallelize a test run across a configurable number of worker processes. While this can speed up CPU-bound test runs, it is mainly useful for IO-bound tests that spend most of their time waiting for data to arrive from someplace else and can benefit from parallelization.

How tests are distributed

The ideal case would be to dispatch each test to a worker process separately, and to have enough worker processes that the entire test run takes only as long as the slowest test. This ideal is not attainable in all cases, however, because many test suites depend on context (class, module or package) fixtures.

Some context fixtures are re-entrant – that is, they can be called many times concurrently. Other context fixtures can be shared among tests running in different processes. Still others must be run once and only once for a given set of tests, and must be in the same process as the tests themselves.

The plugin can't know the difference between these types of context fixtures unless you tell it, so the default behavior is to dispatch the entire context suite to a worker as a unit. This way, the fixtures are run once, in the same process as the tests. (That, of course, is how they are run when the plugin is not active: All tests are run in a single process.)

Controlling distribution

There are two context-level variables that you can use to control this default behavior.

If a context's fixtures are re-entrant, set `_multiprocess_can_split_ = True` in the context, and the plugin will dispatch tests in suites bound to that context as if the context had no fixtures. This means that the fixtures will execute multiple times, typically once per test, and concurrently.

For example, a module that contains re-entrant fixtures might look like:

```
_multiprocess_can_split_ = True

def setup():
    ...
```

A class might look like:

```
class TestClass:
    _multiprocess_can_split_ = True

    @classmethod
    def setup_class(cls):
        ...
```

Alternatively, if a context's fixtures may only be run once, or may not run concurrently, but *may* be shared by tests running in different processes – for instance a package-level fixture that starts an external http server or initializes a shared database – then set `_multiprocess_shared_ = True` in the context. Fixtures for contexts so marked will execute in the primary nose process, and tests in those contexts will be individually dispatched to run in parallel.

A module with shareable fixtures might look like:

```
_multiprocess_shared_ = True

def setup():
    ...
```

A class might look like:

```
class TestClass:
    _multiprocess_shared_ = True

    @classmethod
    def setup_class(cls):
        ...
```

These options are mutually exclusive: you can't mark a context as both splittable and shareable.

Example

Consider three versions of the same test suite. One is marked `_multiprocess_shared_`, another `_multiprocess_can_split_`, and the third is unmarked. They all define the same fixtures:

called = []

def setup(): print "setup called" called.append('setup')

def teardown(): print "teardown called" called.append('teardown')

And each has two tests that just test that `setup()` has been called once and only once.

When run without the multiprocess plugin, fixtures for the shared, can-split and not-shared test suites execute at the same times, and all tests pass.

Note: The run() function in *nose.plugins.plugintest* reformats test result output to remove timings, which will vary from run to run, and redirects the output to stdout.

```
>>> from nose.plugins.plugintest import run_buffered as run
```

```
>>> import os
>>> support = os.path.join(os.path.dirname(__file__), 'support')
>>> test_not_shared = os.path.join(support, 'test_not_shared.py')
>>> test_shared = os.path.join(support, 'test_shared.py')
>>> test_can_split = os.path.join(support, 'test_can_split.py')
```

The module with shared fixtures passes.

```
>>> run(argv=['nosetests', '-v', test_shared])
setup called
test_shared.TestMe.test_one ... ok
test_shared.test_a ... ok
test_shared.test_b ... ok
teardown called

----------------------------------------------------------------------
Ran 3 tests in ...s

OK
```

As does the module with no fixture annotations.

```
>>> run(argv=['nosetests', '-v', test_not_shared])
setup called
test_not_shared.TestMe.test_one ... ok
test_not_shared.test_a ... ok
test_not_shared.test_b ... ok
teardown called

----------------------------------------------------------------------
Ran 3 tests in ...s

OK
```

And the module that marks its fixtures as re-entrant.

```
>>> run(argv=['nosetests', '-v', test_can_split])
setup called
test_can_split.TestMe.test_one ... ok
test_can_split.test_a ... ok
test_can_split.test_b ... ok
teardown called
```

```
----------------------------------------------------------------------
Ran 3 tests in ...s

OK
```

However, when run with the `--processes=2` switch, each test module behaves differently.

```
>>> from nose.plugins.multiprocess import MultiProcess
```

The module marked `_multiprocess_shared_` executes correctly, although as with any use of the multiprocess plugin, the order in which the tests execute is indeterminate.

First we have to reset all of the test modules.

```
>>> import sys
>>> sys.modules['test_not_shared'].called[:] = []
>>> sys.modules['test_can_split'].called[:] = []
```

Then we can run the tests again with the multiprocess plugin active.

```
>>> run(argv=['nosetests', '-v', '--processes=2', test_shared],
...     plugins=[MultiProcess()])
setup called
test_shared.... ok
teardown called

----------------------------------------------------------------------
Ran 3 tests in ...s

OK
```

As does the one not marked – however in this case, `--processes=2` will do *nothing at all*: since the tests are in a module with unmarked fixtures, the entire test module will be dispatched to a single runner process.

However, the module marked `_multiprocess_can_split_` will fail, since the fixtures *are not reentrant*. A module such as this *must not* be marked `_multiprocess_can_split_`, or tests will fail in one or more runner processes as fixtures are re-executed.

We have to reset all of the test modules again.

```
>>> import sys
>>> sys.modules['test_not_shared'].called[:] = []
>>> sys.modules['test_can_split'].called[:] = []
```

Then we can run again and see the failures.

```
>>> run(argv=['nosetests', '-v', '--processes=2', test_can_split],
...     plugins=[MultiProcess()])
setup called
teardown called
...
test_can_split....
...
FAILED (failures=...)
```

Other differences in test running

The main difference between using the multiprocess plugin and not doing so is obviously that tests run concurrently under multiprocess. However, there are a few other differences that may impact your test suite:

- More tests may be found

  Because tests are dispatched to worker processes by name, a worker process may find and run tests in a module that would not be found during a normal test run. For instance, if a non-test module contains a test-like function, that function would be discovered as a test in a worker process if the entire module is dispatched to the worker. This is because worker processes load tests in *directed* mode – the same way that nose loads tests when you explicitly name a module – rather than in *discovered* mode, the mode nose uses when looking for tests in a directory.

- Out-of-order output

  Test results are collected by workers and returned to the master process for output. Since different processes may complete their tests at different times, test result output order is not determinate.

- Plugin interaction warning

  The multiprocess plugin does not work well with other plugins that expect to wrap or gain control of the test-running process. Examples from nose's builtin plugins include coverage and profiling: a test run using both multiprocess and either of those is likely to fail in some confusing and spectacular way.

- Python 2.6 warning

  This is unlikely to impact you unless you are writing tests for nose itself, but be aware that under python 2.6, the multiprocess plugin is not re-entrant. For example, when running nose with the plugin active, you can't use subprocess to launch another copy of nose that also uses the multiprocess plugin. This is why this test is skipped under python 2.6 when run with the `--processes` switch.

Restricted Plugin Managers

In some cases, such as running under the `python setup.py test` command, nose is not able to use all available plugins. In those cases, a *nose.plugins.manager.RestrictedPluginManager* is used to exclude plugins that implement API methods that nose is unable to call.

Support files for this test are in the support directory.

```
>>> import os
>>> support = os.path.join(os.path.dirname(__file__), 'support')
```

For this test, we'll use a simple plugin that implements the `startTest` method.

```
>>> from nose.plugins.base import Plugin
>>> from nose.plugins.manager import RestrictedPluginManager
>>> class StartPlugin(Plugin):
...     def startTest(self, test):
...         print "started %s" % test
```

Note: The run() function in *nose.plugins.plugintest* reformats test result output to remove timings, which will vary from run to run, and redirects the output to stdout.

```
>>> from nose.plugins.plugintest import run_buffered as run
```

When run with a normal plugin manager, the plugin executes.

```
>>> argv = ['plugintest', '-v', '--with-startplugin', support]
>>> run(argv=argv, plugins=[StartPlugin()])
started test.test
test.test ... ok

----------------------------------------------------------------------
Ran 1 test in ...s

OK
```

However, when run with a restricted plugin manager configured to exclude plugins implementing *startTest*, an exception is raised and nose exits.

```
>>> restricted = RestrictedPluginManager(
...     plugins=[StartPlugin()], exclude=('startTest',), load=False)
>>> run(argv=argv, plugins=restricted)
Traceback (most recent call last):
...
SystemExit: 2
```

Errors are only raised when options defined by excluded plugins are used.

```
>>> argv = ['plugintest', '-v', support]
>>> run(argv=argv, plugins=restricted)
test.test ... ok

----------------------------------------------------------------------
Ran 1 test in ...s

OK
```

When a disabled option appears in a configuration file, instead of on the command line, a warning is raised instead of an exception.

```
>>> argv = ['plugintest', '-v', '-c', os.path.join(support, 'start.cfg'),
...         support]
>>> run(argv=argv, plugins=restricted)
RuntimeWarning: Option 'with-startplugin' in config file '...start.cfg' ignored:␣
↪excluded by runtime environment
test.test ... ok

----------------------------------------------------------------------
Ran 1 test in ...s

OK
```

However, if an option appears in a configuration file that is not recognized either as an option defined by nose, or by an active or excluded plugin, an error is raised.

```
>>> argv = ['plugintest', '-v', '-c', os.path.join(support, 'bad.cfg'),
...         support]
>>> run(argv=argv, plugins=restricted)
Traceback (most recent call last):
...
SystemExit: 2
```

Using a Custom Selector

By default, nose uses a *nose.selector.Selector* instance to decide what is and is not a test. The default selector is fairly simple: for the most part, if an object's name matches the `testMatch` regular expression defined in the active *nose.config.Config* instance, the object is selected as a test.

This behavior is fine for new projects, but may be undesireable for older projects with a different test naming scheme. Fortunately, you can easily override this behavior by providing a custom selector using a plugin.

```
>>> import os
>>> support = os.path.join(os.path.dirname(__file__), 'support')
```

In this example, the project to be tested consists of a module and package and associated tests, laid out like this:

```
>>> from nose.util import ls_tree
>>> print ls_tree(support)
|-- mymodule.py
|-- mypackage
|   |-- __init__.py
|   |-- strings.py
|   `-- math
|       |-- __init__.py
|       `-- basic.py
`-- tests
    |-- testlib.py
    |-- math
    |   `-- basic.py
    |-- mymodule
    |   `-- my_function.py
    `-- strings
        `-- cat.py
```

Because the test modules do not include `test` in their names, nose's default selector is unable to discover this project's tests.

Note: The run() function in *nose.plugins.plugintest* reformats test result output to remove timings, which will vary from run to run, and redirects the output to stdout.

```
>>> from nose.plugins.plugintest import run_buffered as run
```

```
>>> argv = [__file__, '-v', support]
>>> run(argv=argv)
----------------------------------------------------------------------
Ran 0 tests in ...s

OK
```

The tests for the example project follow a few basic conventions:

- The are all located under the tests/ directory.
- Test modules are organized into groups under directories named for the module or package they test.
- testlib is *not* a test module, but it must be importable by the test modules.

- Test modules contain unitest.TestCase classes that are tests, and may contain other functions or classes that are NOT tests, no matter how they are named.

We can codify those conventions in a selector class.

```
>>> from nose.selector import Selector
>>> import unittest
>>> class MySelector(Selector):
...     def wantDirectory(self, dirname):
...         # we want the tests directory and all directories
...         # beneath it, and no others
...         parts = dirname.split(os.path.sep)
...         return 'tests' in parts
...     def wantFile(self, filename):
...         # we want python modules under tests/, except testlib
...         parts = filename.split(os.path.sep)
...         base, ext = os.path.splitext(parts[-1])
...         return 'tests' in parts and ext == '.py' and base != 'testlib'
...     def wantModule(self, module):
...         # wantDirectory and wantFile above will ensure that
...         # we never see an unwanted module
...         return True
...     def wantFunction(self, function):
...         # never collect functions
...         return False
...     def wantClass(self, cls):
...         # only collect TestCase subclasses
...         return issubclass(cls, unittest.TestCase)
```

To use our selector class, we need a plugin that can inject it into the test loader.

```
>>> from nose.plugins import Plugin
>>> class UseMySelector(Plugin):
...     enabled = True
...     def configure(self, options, conf):
...         pass # always on
...     def prepareTestLoader(self, loader):
...         loader.selector = MySelector(loader.config)
```

Now we can execute a test run using the custom selector, and the project's tests will be collected.

```
>>> run(argv=argv, plugins=[UseMySelector()])
test_add (basic.TestBasicMath) ... ok
test_sub (basic.TestBasicMath) ... ok
test_tuple_groups (my_function.MyFunction) ... ok
test_cat (cat.StringsCat) ... ok

----------------------------------------------------------------------
Ran 4 tests in ...s

OK
```

Finding tests in all modules

Normally, nose only looks for tests in modules whose names match testMatch. By default that means modules with 'test' or 'Test' at the start of the name after an underscore (_) or dash (-) or other non-alphanumeric character.

If you want to collect tests from all modules, use the `--all-modules` command line argument to activate the *allmodules plugin*.

Note: The function `nose.plugins.plugintest.run()` reformats test result output to remove timings, which will vary from run to run, and redirects the output to stdout.

```
>>> from nose.plugins.plugintest import run_buffered as run
```

```
>>> import os
>>> support = os.path.join(os.path.dirname(__file__), 'support')
>>> argv = [__file__, '-v', support]
```

The target directory contains a test module and a normal module.

```
>>> support_files = [d for d in os.listdir(support)
...                  if not d.startswith('.')
...                  and d.endswith('.py')]
>>> support_files.sort()
>>> support_files
['mod.py', 'test.py']
```

When run without `--all-modules`, only the test module is examined for tests.

```
>>> run(argv=argv)
test.test ... ok

----------------------------------------------------------------------
Ran 1 test in ...s

OK
```

When `--all-modules` is active, both modules are examined.

```
>>> from nose.plugins.allmodules import AllModules
>>> argv = [__file__, '-v', '--all-modules', support]
>>> run(argv=argv, plugins=[AllModules()])
mod.test ... ok
mod.test_fails ... FAIL
test.test ... ok

======================================================================
FAIL: mod.test_fails
----------------------------------------------------------------------
Traceback (most recent call last):
...
AssertionError: This test fails

----------------------------------------------------------------------
Ran 3 tests in ...s

FAILED (failures=1)
```

XUnit output supports skips

```
>>> import os
>>> from nose.plugins.xunit import Xunit
>>> from nose.plugins.skip import SkipTest, Skip
>>> support = os.path.join(os.path.dirname(__file__), 'support')
>>> outfile = os.path.join(support, 'nosetests.xml')
>>> from nose.plugins.plugintest import run_buffered as run
>>> argv = [__file__, '-v', '--with-xunit', support,
...         '--xunit-file=%s' % outfile]
>>> run(argv=argv, plugins=[Xunit(), Skip()])
test_skip.test_ok ... ok
test_skip.test_err ... ERROR
test_skip.test_fail ... FAIL
test_skip.test_skip ... SKIP: not me

======================================================================
ERROR: test_skip.test_err
----------------------------------------------------------------------
Traceback (most recent call last):
...
Exception: oh no

======================================================================
FAIL: test_skip.test_fail
----------------------------------------------------------------------
Traceback (most recent call last):
...
AssertionError: bye

----------------------------------------------------------------------
XML: ...nosetests.xml
----------------------------------------------------------------------
Ran 4 tests in ...s

FAILED (SKIP=1, errors=1, failures=1)
```

```
>>> result_file = open(outfile, 'r')
>>> result_file.read()
'<?xml version="1.0" encoding="UTF-8"?><testsuite name="nosetests" tests="4" errors="1
↪" failures="1" skip="1"><testcase classname="test_skip" name="test_ok" time="..."></
↪testcase><testcase classname="test_skip" name="test_err" time="..."><error type="...
↪Exception" message="oh no">...</error></testcase><testcase classname="test_skip"␣
↪name="test_fail" time="..."><failure type="...AssertionError" message="bye">...</
↪failure></testcase><testcase classname="test_skip" name="test_skip" time="...">
↪<skipped type="...SkipTest" message="not me">...</skipped></testcase></testsuite>'
>>> result_file.close()
```

About the name

- nose is the least silly short synonym for discover in the dictionary.com thesaurus that does not contain the word 'spy.'
- Pythons have noses
- The nose knows where to find your tests

- Nose Obviates Suite Employment

Contact the author

You can email me at jpellerin+nose at gmail dot com.

To report bugs, ask questions, or request features, please use the *issues* tab at the Google code site: http://code.google.com/p/python-nose/issues/list. Patches are welcome!

Similar test runners

nose was inspired mainly by py.test, which is a great test runner, but formerly was not all that easy to install, and is not based on unittest.

Test suites written for use with nose should work equally well with py.test, and vice versa, except for the differences in output capture and command line arguments for the respective tools.

License and copyright

Articles, etc

- An Extended Introduction to the nose Unit Testing Framework: Titus Brown's excellent article provides a great overview of nose and its uses.
- My blog
- Tweets

Python Module Index

n

Index

Symbols

A

B

C

D

E

F

G

H

O

P

R

S

T

U

V

W

X