nose2 Documentation

Release 0.4.7

Jason Pellerin

August 13, 2013

CONTENTS

nose2 is the next generation of nicer testing for Python, based on the plugins branch of unittest2. nose2 aims to improve on nose by:

- providing a better plugin api
- being easier for users to configure
- simplifying internal interfaces and processes
- supporting Python 2 and 3 from the same codebase, without translation
- encourging greater community involvment in its development

In service of some those goals, some features of nose *will not* be supported in nose2. See differences for a thorough rundown.

In time – once unittest2 supports plugins – nose2 should be able to become just a collection of plugins and configuration defaults. For now, it provides a plugin api similar to the one in the unittest2 plugins branch, and overrides various unittest2 objects.

You are witnesses at the new birth of nose, mark 2. Hope you enjoy our new direction!

CHAPTER

ONE

USER'S GUIDE

1.1 Getting started with nose2

1.1.1 Installation

The recommended way to install nose2 is with pip

```
pip install nose2
```

You can also install from source by downloading the source distribution from pypi, un-taring it, and running `python setup.py install` in the source directory. Note that if you install this way, and do not have distribute or setuptools installed, you must install nose2's dependencies manually.

Dependencies

For Python 2.7, Python 3.2 and pypy, nose2 requires six version 1.1. For Python 2.6, nose2 also requires argparse version 1.2.1 and unittest2 version 0.5.1. When installing with pip, distribute or setuptools, these dependencies will be installed automatically.

Development version

You can install the development version of nose2 from github with pip:

```
pip install -e git+git://github.com/nose-devs/nose2.git#egg=nose2
```

You can also download a package from github, or clone the source and install from there with `python setup.py install`.

1.1.2 Running tests

To run tests in a project, use the `nose2` script that is installed with nose2:

```
nose2
```

This will find and run tests in all packages in the current working directory, and any sub-directories of the current working directory whose names start with 'test'.

To find tests, nose2 looks for modules whose names start with 'test'. In those modules, nose2 will load tests from all `unittest.TestCase` subclasses, as well as functions whose names start with 'test'.

The `nose2` script supports a number of command-line options, as well as extensive configuration via config files. For more information see *Using nose2* and *Configuring nose2*.

1.2 Using nose2

1.2.1 Running Tests

In the simplest case, go to the directory that includes your project source and run `nose2` there:

```
nose2
```

This will discover tests in packages and test directories under that directory, load them, and run them, then output something like:

```
.....................................................................
----------------------------------------------------------------------
Ran 77 tests in 1.897s

OK
```

"Test directories" means any directories whose names start with "test". Within test directories and within any Python packages found in the starting directory and any source directories in the starting directory, nose2 will discover test modules and load tests from them. "Test modules" means any modules whose names start with "test".

Within test modules, nose2 will load tests from `unittest.TestCase` subclasses, and from test functions (functions whose names begin with "test").

To change the place discovery starts, or to change the top-level importable directory of the project, use the `-s` and `-t` options.

-s START_DIR, **-start-dir** START_DIR
: Directory to start discovery. Defaults to the current working directory. This directory is where nose2 will start looking for tests.

-t TOP_LEVEL_DIRECTORY, **-top-level-directory** TOP_LEVEL_DIRECTORY, **-project-directory** TOP_LE
: Top-level directory of the project. Defaults to the starting directory. This is the directory containing importable modules and packages, and is always prepended to sys.path before test discovery begins.

Specifying Tests to Run

Pass *test names* to nose2 on the command line to run individual test modules, classes, or tests.

A test name consists of a *python object part* and, for generator or parameterized tests, an *argument part*. The *python object part* is a dotted name, such as `pkg1.tests.test_things.SomeTests.test_ok`. The argument part is separated from the python object part by a colon (":") and specifies the *index* of the generated test to select, *starting from 1*. For example, `pkg1.test.test_things.test_params_func:1` would select the *first* test generated from the parameterized test `test_params_func`.

Plugins may provide other means of test selection.

Running Tests with `python setup.py test`

nose2 supports distribute/setuptools' `python setup.py test` standard for running tests. To use nose2 to run your package's tests, add the following to your setup.py:

```
setup(...
      test_suite='nose2.collector.collector',
      ...
      )
```

(Not literally. Don't put the '...' parts in.)

Two warnings about running tests this way.

One: because the setuptools test command is limited, nose2 returns a "test suite" that actually takes over the test running process completely, bypassing the test result and test runner that call it. This may be incompatible with some packages.

Two: because the command line arguments to the test command may not match up properly with nose2's arguments, the nose2 instance started by the collector *does not accept any command line arguments*. This means that it always runs all tests, and that you cannot configure plugins on the command line when running tests this way. As a workaround, when running under the test command, nose2 will read configuration from `setup.cfg` if it is present, in addition to `unittest.cfg` and `nose2.cfg`. This enables you to put configuration specific to the setuptools test command in `setup.cfg` – for instance to activate plugins that you would otherwise activate via the command line.

1.2.2 Getting Help

Run:

```
nose2 -h
```

to get help for nose2 itself and all loaded plugins.

```
usage: nose2 [-s START_DIR] [-t TOP_LEVEL_DIRECTORY] [--config [CONFIG]]
             [--no-user-config] [--no-plugins] [--verbose] [--quiet] [-B] [-D]
             [--collect-only] [--log-capture] [-P] [-h]
             [testNames [testNames ...]]

positional arguments:
  testNames

optional arguments:
  -s START_DIR, --start-dir START_DIR
                        Directory to start discovery ('.' default)
  -t TOP_LEVEL_DIRECTORY, --top-level-directory TOP_LEVEL_DIRECTORY, --project-directory TOP_LEVEL_DI
                        Top level directory of project (defaults to start dir)
  --config [CONFIG], -c [CONFIG]
                        Config files to load, if they exist. ('unittest.cfg'
                        and 'nose2.cfg' in start directory default)
  --no-user-config      Do not load user config files
  --no-plugins          Do not load any plugins. Warning: nose2 does not do
                        anything if no plugins are loaded
  --verbose, -v
  --quiet
  -h, --help            Show this help message and exit

plugin arguments:
  Command-line arguments added by plugins:

  -B, --output-buffer   Enable output buffer
  -D, --debugger        Enter pdb on test fail or error
  --collect-only        Collect and output test names, do not run any tests
```

```
  --log-capture         Enable log capture
  -P, --print-hooks     Print names of hooks in order of execution
```

1.3 Configuring nose2

1.3.1 Configuration Files

Most configuration of nose2 is done via config files. These are standard, .ini-style config files, with sections marked off by brackets ("`[unittest]`") and `key = value` pairs within those sections.

Two command line options, *-c* and *--no-user-config* may be used to determine which config files are loaded.

-c CONFIG, **-config** CONFIG
: Config files to load. Default behavior is to look for `unittest.cfg` and `nose2.cfg` in the start directory, as well as any user config files (unless *--no-user-config* is selected).

-no-user-config
: Do not load user config files. If not specified, in addition to the standard config files and any specified with *-c*, nose2 will look for `.unittest.cfg` and `.nose2.cfg` in the user's $HOME directory.

Configuring Test Discovery

The `[unittest]` section of nose2 config files is used to configure nose2 itself. The following options are available to configure test discovery:

start-dir
: This option configures the default directory to start discovery. The default value is "." (the current directory where nose2 is executed). This directory is where nose2 will start looking for tests.

code-directories
: This option configures nose2 to add the named directories to sys.path and the discovery path. Use this if your project has code in a location other than the top level of the project, or the directories `lib` or `src`. The value here may be a list: put each directory on its own line in the config file.

test-file-pattern
: This option configures how nose detects test modules. It is a file glob.

test-method-prefix
: This option configures how nose detects test functions and methods. The prefix set here will be matched (via simple string matching) against the start of the name of each method in test cases and each function in test modules.

Examples:

```
[unittest]
start-dir = tests
code-directories = source
                   more_source
test-file-pattern = *_test.py
test-method-prefix = t
```

Specifying Plugins to Load

To avoid loading any plugins, use the *--no-plugins* option. Beware, though: nose2 does all test discovery and loading via plugins, so unless you are patching in a custom test loader and runner, when run with *--no-plugins*,

nose2 will do nothing.

-no-plugins
Do not load any plugins. *This kills the nose2.*

To specify plugins to load beyond the builtin plugins automatically loaded, add a `plugins` entry under the `[unittest]` section in a config file.

plugins
List of plugins to load. Put one plugin module on each line.

To exclude some plugins that would otherwise be loaded, add an `exclude-plugins` entry under the `[unittest]` section in a config file.

exclude-plugins
List of plugins to exclude. Put one plugin module on each line.

Note: It bears repeating that in both `plugins` and `exclude-plugins` entries, you specify the plugin *module*, not the plugin *class*.

Examples:

```
[unittest]
plugins = myproject.plugins.frobulate
          otherproject.contrib.plugins.derper

exclude-plugins = nose2.plugins.loader.functions
                  nose2.plugins.outcomes
```

1.3.2 Configuring Plugins

Most plugins specify a config file section that may be used to configure the plugin. If nothing else, any plugin that specifies a config file section can be set to automatically register by including `always-on = True` in its config:

```
[my-plugin]
always-on = True
```

Plugins may accept any number of other config values, which may be booleans, strings, integers or lists. A polite plugin will document these options somewhere. Plugins that want to make use of nose2's Sphinx extension as detailed in *Documenting plugins* *must* extract all of their config values in their `__init__` methods.

1.3.3 Test Runner Tips and Tweaks

Running Tests in a Single Module

You can use `nose2.main` in the same way that `unittest.main` (and `unittest2.main`) have historically worked: to run the tests in a single module. Just put a block like the following at the end of the module:

```
if __name__ == '__main__':
    import nose2
    nose2.main()
```

Then *run the module directly* – In other words, do not run the `nose2` script.

Rolling Your Own Runner

You can take more control over the test runner by foregoing the `nose2` script and rolling your own. To do that, you just need to write a script that calls `nose2.discover`, for instance:

```
if __name__ == '__main__':
  import nose2
  nose2.discover()
```

You can pass several keyword arguments to `nose2.discover`, all of which are detailed in the documentation for `nose2.main.PluggableTestProgram`.

Altering the Default Plugin Set

To add plugin *modules* to the list of those automatically loaded, you can pass a list of module names to add (the `plugins`) argument or exclude (`excludedPlugins`). You can also subclass `nose2.main.PluggableTestProgram` and set the class-level `defaultPlugins` and `excludePlugins` attributes to alter plugin loading.

When Loading Plugins from Modules is not Enough

None of which will help if you need to register a plugin *instance* that you've loaded yourself. For that, use the `extraHooks` keyword argument to `nose2.discover`. Here, you pass in a list of 2-tuples, each of which contains a hook name and a plugin *instance* to register for that hook. This allows you to register plugins that need runtime configuration that is not easily passed in through normal channels – and also to register *objects that are not nose2 plugins* as hook targets. Here's a trivial example:

```
if __name__ == '__main__':
  import nose2

  class Hello(object):
      def startTestRun(self, event):
          print("hello!")

  nose2.discover(extraHooks=[('startTestRun', Hello())])
```

This can come in handy when integrating with other systems that expect you to provide a test runner that they execute, rather than executing tests yourself (django, for instance).

1.4 Differences: nose2 vs nose vs unittest2

1.4.1 nose2 is not nose

What's Different

Python Versions

nose supports Python 2.4 and above, but nose2 *only supports Python 2.6, 2.7, 3.2, 3.3 and pypy*. Unfortunately, supporting Pythons older than 2.6 along with Python 3 in the same codebase is not practical. Since that is one of the core goals of nose2, support for older versions of Python had to be sacrificed.

Test Discovery and Loading

nose loads test modules lazily: tests in the first-loaded module are executed before the second module is imported. *nose2 loads all tests first, then begins test execution*. This has some important implications.

First, it means that nose2 does not need a custom importer. nose2 imports test modules with `__import__()`.

Second, it means that *nose2 does not support all of the test project layouts that nose does*. Specifically, projects that look like this will fail to load tests correctly with nose2:

```
.
`-- tests
    |-- more_tests
    |   `-- test.py
    `-- test.py
```

To nose's loader, those two test modules look like different modules. But to nose2's loader, they look the same, and will not load correctly.

Test Fixtures

nose2 supports only the *same levels of fixtures as unittest2*. This means class level fixtures and module level fixtures are supported, but *package-level fixtures are not*. In addition, unlike nose, nose2 does not attempt to order tests named on the command-line to group those with the same fixtures together.

Parameterized and Generator Tests

nose2 supports *more kinds of parameterized and generator tests than nose*, and supports all test generators in test functions, test classes, and in unittest TestCase subclasses. nose supports them only in test functions and test classes that do not subclass unittest.TestCase. See: *Loader: Test Generators* and *Loader: Parameterized Tests* for more.

Configuration

nose expects plugins to make all of their configuration parameters available as command-line options. *nose2 expects almost all configuration to be done via configuration files*. Plugins should generally have only one command-line option: the option to activate the plugin. Other configuration parameters should be loaded from config files. This allows more repeatable test runs and keeps the set of command-line options small enough for humans to read. See: *Configuring nose2* for more.

Plugin Loading

nose uses setuptools entry points to find and load plugins. nose2 does not. Instead, *nose2 requires that all plugins be listed in config files*. This ensures that no plugin is loaded into a test system just by virtue of being installed somewhere, and makes it easier to include plugins that are part of the project under test. See: *Configuring nose2* for more.

Limited support for `python setup.py test`

nose2 supports setuptools' `python setup.test` command, but via very different means than nose. To avoid the internal complexity forced on nose by the fact that the setuptools test command can't be configured with a custom test runner, when run this way, *nose2 essentially hijacks the test running process*. The "test suite" that `nose2.collector.collector()` returns actually *is* a test runner, cloaked inside of a test case. It loads and

runs tests as normal, setting up its own test runner and test result, and calls sys.exit() itself – completely bypassing the test runner and test result that setuptools/unittest create. This may be incompatible with some projects.

New Plugin API

nose2 implements a new plugin API based on the work done by Michael Foord in unittest2's plugins branch. This API is greatly superior to the one in nose, especially in how it allows plugins to interact with each other. But it is different enough from the API in nose that supporting nose plugins in nose2 will not be practical: *plugins must be rewritten to work with nose2*. See: *Writing Plugins* for more.

Missing Plugins

nose2 does not (yet) include some of the more commonly-used plugins in nose. Most of these should arrive in future releases. However, some of nose's builtin plugins cannot be ported to nose2 due to differences in internals. See: *Plugins for nose2* for information on the plugins built in to nose2.

Internals

nose wraps or replaces everything in unittest. nose2 does a bit less: *it does not wrap TestCases*, and does not wrap the test result class with a result proxy. nose2 does subclass TestProgram, and install its own loader, runner and result classes. It does this unconditionally, rather than allowing arguments to `TestProgram.__init__()` to specify the test loader and runner. See *Internals* for more information.

License

While nose was LGPL, nose2 is BSD licensed. This change was made at the request of the majority of nose contributors.

What's the Same

Philosophy

nose2 has the same goals as nose: to extend unittest to make testing nicer and easier to understand. It aims to give developers flexibility, power and transparency, so that common test scenarios require no extra work, and uncommon test scenarios can be supported with minimal fuss and magic.

People

nose2 is being developed by the same people who maintain nose.

1.4.2 nose2 is not (exactly) unittest2/plugins

nose2 is based on the unittest2 plugins branch, but differs from it in several substantial ways. The *event api not exactly the same* because nose2 can't replace unittest.TestCase, and *does not configure the test run or plugin set globally*. nose2 also has a *wholly different reporting API* from unittest2's plugins, one which we feel better supports some common cases (like adding extra information to error output). nose2 also *defers more work to plugins* than unittest2: the test loader, runner and result are just plugin callers, and all of the logic of test discovery, running and reporting is

implemented in plugins. This means that unlike unittest2, *nose2 includes a substantial set of plugins that are active by default*.

1.5 Plugins for nose2

1.5.1 Built in and Loaded by Default

These plugins are loaded by default. To exclude one of these plugins from loading, add the plugin's module name to the `exclude-plugins` list in a config file's `[unittest]` section, or pass the plugin module with the `--exclude-plugin` argument on the command line. You can also pass plugin module names to exclude to a `nose2.main.PluggableTestProgram` using the `excludePlugins` keyword argument.

Loader: Test discovery

Discovery-based test loader.

This plugin implements nose2's automatic test module discovery. It looks for test modules in packages and directories whose names start with 'test', then fires the `loadTestsFromModule()` hook for each one to allow other plugins to load the actual tests.

It also fires `handleFile()` for every file that it sees, and `matchPath()` for every python module, to allow other plugins to load tests from other kinds of files and to influence which modules are examined for tests.

Configuration [discovery]

always-on

Default True

Type boolean

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[discovery]
always-on = True
```

Plugin class reference: DiscoveryLoader

class nose2.plugins.loader.discovery.**DiscoveryLoader**

Loader plugin that can discover tests

loadTestsFromName(*event*)

Load tests from module named by event.name

loadTestsFromNames(*event*)

Discover tests if no test names specified

Loader: Test Functions

Load tests from test functions in modules.

This plugin responds to `loadTestsFromModule()` by adding test cases for all test functions in the module to `event.extraTests`. It uses `session.testMethodPrefix` to find test functions.

Functions that are generators, have param lists, or take arguments are not collected.

This plugin also implements `loadTestsFromName()` to enable loading tests from dotted function names passed on the command line.

Configuration [functions]

always-on

Default True

Type boolean

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[functions]
always-on = True
```

Plugin class reference: Functions

class nose2.plugins.loader.functions.**Functions**
Loader plugin that loads test functions

loadTestsFromModule(*event*)
Load test functions from event.module

loadTestsFromName(*event*)
Load test if event.name is the name of a test function

Loader: Test Generators

Load tests from generators.

This plugin implements `loadTestFromTestCase()`, `loadTestsFromName()` and `loadTestFromModule()` to enable loading tests from generators.

Generators may be functions or methods in test cases. In either case, they must yield a callable and arguments for that callable once for each test they generate. The callable and arguments may all be in one tuple, or the arguments may be grouped into a separate tuple:

```
def test_gen():
    yield check, 1, 2
    yield check, (1, 2)
```

To address a particular generated test via a command-line test name, append a colon (':') followed by the index, *starting from 1*, of the generated case you want to execute.

Configuration [generators]

always-on

Default True

Type boolean

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[generators]
always-on = True
```

Plugin class reference: Generators

class nose2.plugins.loader.generators.**Generators**
Loader plugin that loads generator tests

getTestCaseNames(*event*)
Get generator test case names from test case class

loadTestsFromModule(*event*)
Load tests from generator functions in a module

loadTestsFromName(*event*)
Load tests from generator named on command line

loadTestsFromTestCase(*event*)
Load generator tests from test case

Loader: Parameterized Tests

Load tests from parameterized functions and methods.

This plugin implements `getTestCaseNames()`, `loadTestsFromModule()`, and `loadTestsFromName()` to support loading tests from parameterized test functions and methods.

To parameterize a function or test case method, use `nose2.tools.params()`.

To address a particular parameterized test via a command-line test name, append a colon (':') followed by the index, *starting from 1*, of the case you want to execute.

Configuration [parameters]

always-on

Default True

Type boolean

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[parameters]
always-on = True
```

Plugin class reference: Parameters

class nose2.plugins.loader.parameters.**Parameters**
Loader plugin that loads parameterized tests

getTestCaseNames(*event*)
Generate test case names for all parameterized methods

loadTestsFromModule(*event*)
Load tests from parameterized test functions in the module

loadTestsFromName(*event*)
Load parameterized test named on command line

Loader: Test Cases

Load tests from `unittest.TestCase` subclasses.

This plugin implements `loadTestsFromName()` and `loadTestsFromModule()` to load tests from `unittest.TestCase` subclasses found in modules or named on the command line.

Configuration [testcases]

always-on

Default True

Type boolean

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[testcases]
always-on = True
```

Plugin class reference: TestCaseLoader

class nose2.plugins.loader.testcases.**TestCaseLoader**
Loader plugin that loads from test cases

loadTestsFromModule(*event*)
Load tests in `unittest.TestCase` subclasses

loadTestsFromName(*event*)
Load tests from event.name if it names a test case/method

Loader: Test Classes

Load tests from classes that are *not* `unittest.TestCase` subclasses.

This plugin responds to `loadTestsFromModule()` by adding test cases for test methods found in classes in the module that are *not* sublcasses of `unittest.TestCase`, but whose names (lowercased) match the configured test method prefix.

Test class methods that are generators or have param lists are not loaded here, but by the nose2.plugins.loader.generators.Generators and nose2.plugins.loader.parameters.Parameters plugins.

This plugin also implements loadTestsFromName() to enable loading tests from dotted class and method names passed on the command line.

This plugin makes two additional plugin hooks available for other test loaders to use:

nose2.plugins.loader.testclasses.**loadTestsFromTestClass**(*self*, *event*)

> **Parameters** **event** – A LoadFromTestClassEvent instance
>
> Plugins can use this hook to load tests from a class that is not a unittest.TestCase subclass. To prevent other plugins from loading tests from the test class, set event.handled to True and return a test suite. Plugins can also append tests to event.extraTests – ususally that's what you want to do, since that will allow other plugins to load their tests from the test case as well.

nose2.plugins.loader.testclasses.**getTestMethodNames**(*self*, *event*)

> **Parameters** **event** – A GetTestMethodNamesEvent instance
>
> Plugins can use this hook to limit or extend the list of test case names that will be loaded from a class that is not a unittest.TestCase subclass by the standard nose2 test loader plugins (and other plugins that respect the results of the hook). To force a specific list of names, set event.handled to True and return a list: this exact list will be the only test case names loaded from the test case. Plugins can also extend the list of names by appending test names to event.extraNames, and exclude names by appending test names to event.excludedNames.

About Test Classes

Test classes are classes that look test-like but are not subclasses of unittest.TestCase. Test classes support all of the same test types and fixtures as test cases.

To "look test-like" a class must have a name that, lowercased, matches the configured test method prefix – "test" by default. Test classes must also be able to be instantiated without arguments.

What are they useful for? Mostly the case where a test class can't for some reason subclass unittest.TestCase. Otherwise, test class tests and test cases are functionally equivalent in nose2, and test cases have broader support and all of those helpful *assert** methods – so when in doubt, you should use a unittest.TestCase.

Here's an example of a test class:

```
class TestSomething(object):

    def test(self):
        assert self.something(), "Something failed!"
```

Configuration [test-classes]

always-on

> **Default** True
>
> **Type** boolean

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[test-classes]
always-on = True
```

Plugin class reference: TestClassLoader

class nose2.plugins.loader.testclasses.**TestClassLoader**
Loader plugin that loads test functions

loadTestsFromModule(*event*)
Load test classes from event.module

loadTestsFromName(*event*)
Load tests from event.name if it names a test class/method

pluginsLoaded(*event*)
Install extra hooks

Adds the new plugin hooks:

•loadTestsFromTestClass

•getTestMethodNames

Loader: load_tests protocol

Loader that implements the load_tests protocol.

This plugin implements the load_tests protocol as detailed in the documentation for unittest2.

See the load_tests protocol documentation for more information.

> **Warning:** Test suites using the load_tests protocol do not work correctly with the multiprocess plugin as of nose2 04. This will be fixed in a future release.

Configuration [load_tests]

always-on

Default True

Type boolean

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[load_tests]
always-on = True
```

Plugin class reference: LoadTestsLoader

class nose2.plugins.loader.loadtests.**LoadTestsLoader**
Loader plugin that implements load_tests.

handleDir(*event*)
Run load_tests in packages.

If a package itself matches the test file pattern, run load_tests in its __init__.py, and stop default test discovery for that package.

moduleLoadedSuite(*event*)
Run load_tests in a module.

May add to or filter tests loaded in module.

Reporting test results

Collect and report test results.

This plugin implements the primary user interface for nose2. It collects test outcomes and reports on them to the console, as well as firing several hooks for other plugins to do their own reporting.

This plugin extends standard unittest console reporting slightly by allowing custom report categories. To put events into a custom reporting category, change the event.outcome to whatever you want. Note, however, that customer categories are *not* treated as errors or failures for the purposes of determining whether a test run has succeeded.

Don't disable this plugin unless you a) have another one doing the same job or b) really don't want any test results (and want all test runs to exit(1))

Configuration [test-result]

always-on

Default True

Type boolean

descriptions

Default True

Type boolean

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[test-result]
always-on = True
descriptions = True
```

Plugin class reference: ResultReporter

class nose2.plugins.result.**ResultReporter**
Result plugin that implements standard unittest console reporting

afterTestRun(*event*)
Handle afterTestRun hook

- prints error lists
- prints summary
- fires summary reporting hooks (`beforeErrorList()`, `beforeSummaryReport()`, etc)

startTest (*event*)
Handle startTest hook

- prints test description if verbosity > 1

testOutcome (*event*)
Handle testOutcome hook

- records test outcome in reportCategories
- prints test outcome label
- fires reporting hooks (`reportSuccess()`, `reportFailure()`, etc)

Buffering test output

Buffer stdout and/or stderr during test execution, appending any output to the error reports of failed tests.

This allows you to use print for debugging in tests without making your test runs noisy.

This plugin implements `startTest()`, `stopTest()`, `setTestOutcome()`, `outcomeDetail()`, `beforeInteraction()` and `afterInteraction()` to manage capturing sys.stdout and/or sys.stderr into buffers, attaching the buffered output to test error report detail, and getting out of the way when other plugins want to talk to the user.

Configuration [output-buffer]

always-on

Default False

Type boolean

stderr

Default False

Type boolean

stdout

Default True

Type boolean

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[output-buffer]
always-on = False
stderr = False
stdout = True
```

Command-line options

-B DEFAULT, **--output-buffer** DEFAULT
Enable output buffer

Plugin class reference: OutputBufferPlugin

class nose2.plugins.buffer.**OutputBufferPlugin**
Buffer output during test execution

afterInteraction(*event*)
Start buffering again (does not clear buffers)

beforeInteraction(*event*)
Stop buffering so users can see stdout

outcomeDetail(*event*)
Add buffered output to event.extraDetail

setTestOutcome(*event*)
Attach buffer(s) to event.metadata

startTest(*event*)
Start buffering selected stream(s)

stopTest(*event*)
Stop buffering

Dropping Into the Debugger

Start a `pdb.post_mortem()` on errors and failures.

This plugin implements `testOutcome()` and will drop into pdb whenever it sees a test outcome that includes exc_info.

It fires `beforeInteraction()` before launching pdb and `afterInteraction()` after. Other plugins may implement `beforeInteraction()` to return False and set event.handled to prevent this plugin from launching pdb.

Configuration [debugger]

always-on

Default False

Type boolean

errors-only

Default False

Type boolean

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[debugger]
always-on = False
errors-only = False
```

Command-line options

-D DEFAULT, **-debugger** DEFAULT
Enter pdb on test fail or error

Plugin class reference: Debugger

class nose2.plugins.debugger.**Debugger**
Enter pdb on test error or failure

pdb
For ease of mocking and using different pdb implementations, pdb is aliased as a class attribute.

pdb = <module 'pdb' from '/usr/lib/python2.7/pdb.pyc'>

testOutcome(*event*)
Drop into pdb on unexpected errors or failures

Stopping After the First Error or Failure

Stop the test run after the first error or failure.

This plugin implements `testOutcome()` and sets `event.result.shouldStop` if it sees an outcome with exc_info that is not expected.

Command-line options

-F DEFAULT, **-fail-fast** DEFAULT
Stop the test run after the first error or failure

Plugin class reference: FailFast

class nose2.plugins.failfast.**FailFast**
Stop the test run after error or failure

testOutcome(*event*)
Stop on unexpected error or failure

Capturing log messages

Capture log messages during test execution, appending them to the error reports of failed tests.

This plugin implements `startTestRun()`, `startTest()`, `stopTest()`, `setTestOutcome()`, and `outcomeDetail()` to set up a logging configuration that captures log messages during test execution, and appends them to error reports for tests that fail or raise exceptions.

Configuration [log-capture]

always-on

Default False

Type boolean

clear-handlers

Default False

Type boolean

date-format

Default None

Type str

filter

Default ['-nose']

Type list

format

Default %(name)s: %(levelname)s: %(message)s

Type str

log-level

Default NOTSET

Type str

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[log-capture]
always-on = False
clear-handlers = False
date-format = None
filter = -nose
format = %(name)s: %(levelname)s: %(message)s
log-level = NOTSET
```

Command-line options

-log-capture DEFAULT
Enable log capture

Plugin class reference: LogCapture

class nose2.plugins.logcapture.**LogCapture**
Capture log messages during test execution

outcomeDetail(*event*)
Append captured log messages to `event.extraDetail`

setTestOutcome(*event*)
Store captured log messages in `event.metadata`

startTest(*event*)
Set up handler for new test

startTestRun(*event*)
Set up logging handler

stopTest (*event*)
Clear captured messages, ready for next test

1.5.2 Built in but *not* Loaded by Default

These plugins are available as part of the nose2 package but *are not loaded by default*. To load one of these plugins, add the plugin module name to the `plugins` list in a config file's `[unittest]` section, or pass the plugin module with the `--plugin` argument on the command line. You can also pass plugin module names to a `nose2.main.PluggableTestProgram` using the `plugins` keyword argument.

Outputting XML Test Reports

Note: New in version 0.2

Output test reports in junit-xml format.

This plugin implements `startTest()`, `testOutcome()` and `stopTestRun()` to compile and then output a test report in junit-xml format. By default, the report is written to a file called `nose2-junit.xml` in the current working directory. You can configure the output filename by setting `path` in a `[junit-xml]` section in a config file. Unicode characters which are invalid in XML 1.0 are replaced with the U+FFFD replacement character. In the case that your software throws an error with an invalid byte string. By default, the ranges of discouraged characters are replaced as well. This can be changed by setting the keep_restricted configuration variable to True.

Configuration [junit-xml]

always-on

Default False

Type boolean

keep_restricted

Default False

Type boolean

path

Default nose2-junit.xml

Type str

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[junit-xml]
always-on = False
keep_restricted = False
path = nose2-junit.xml
```

Command-line options

-X DEFAULT, **-junit-xml** DEFAULT
Generate junit-xml output report

Plugin class reference: JUnitXmlReporter

class nose2.plugins.junitxml.**JUnitXmlReporter**

Output junit-xml test report to file

startTest(*event*)

Count test, record start time

stopTestRun(*event*)

Output xml tree to file

testOutcome(*event*)

Add test outcome to xml tree

Sample output

The XML test report for nose2's sample scenario with tests in a package looks like this:

```
<testsuite errors="1" failures="5" name="nose2-junit" skips="1" tests="25" time="0.004">
  <testcase classname="pkg1.test.test_things" name="test_gen:1" time="0.000141" />
  <testcase classname="pkg1.test.test_things" name="test_gen:2" time="0.000093" />
  <testcase classname="pkg1.test.test_things" name="test_gen:3" time="0.000086" />
  <testcase classname="pkg1.test.test_things" name="test_gen:4" time="0.000086" />
  <testcase classname="pkg1.test.test_things" name="test_gen:5" time="0.000087" />
  <testcase classname="pkg1.test.test_things" name="test_gen_nose_style:1" time="0.000085" />
  <testcase classname="pkg1.test.test_things" name="test_gen_nose_style:2" time="0.000090" />
  <testcase classname="pkg1.test.test_things" name="test_gen_nose_style:3" time="0.000085" />
  <testcase classname="pkg1.test.test_things" name="test_gen_nose_style:4" time="0.000087" />
  <testcase classname="pkg1.test.test_things" name="test_gen_nose_style:5" time="0.000086" />
  <testcase classname="pkg1.test.test_things" name="test_params_func:1" time="0.000093" />
  <testcase classname="pkg1.test.test_things" name="test_params_func:2" time="0.000098">
    <failure message="test failure">Traceback (most recent call last):
  File "nose2/plugins/loader/parameters.py", line 162, in func
    return obj(*argSet)
  File "nose2/tests/functional/support/scenario/tests_in_package/pkg1/test/test_things.py", line 64,
    assert a == 1
AssertionError
</failure>
  </testcase>
  <testcase classname="pkg1.test.test_things" name="test_params_func_multi_arg:1" time="0.000094" />
  <testcase classname="pkg1.test.test_things" name="test_params_func_multi_arg:2" time="0.000089">
    <failure message="test failure">Traceback (most recent call last):
  File "nose2/plugins/loader/parameters.py", line 162, in func
    return obj(*argSet)
  File "nose2/tests/functional/support/scenario/tests_in_package/pkg1/test/test_things.py", line 69,
    assert a == b
AssertionError
</failure>
  </testcase>
  <testcase classname="pkg1.test.test_things" name="test_params_func_multi_arg:3" time="0.000096" />
  <testcase classname="" name="test_fixt" time="0.000091" />
  <testcase classname="" name="test_func" time="0.000084" />
  <testcase classname="pkg1.test.test_things.SomeTests" name="test_failed" time="0.000113">
    <failure message="test failure">Traceback (most recent call last):
  File "nose2/tests/functional/support/scenario/tests_in_package/pkg1/test/test_things.py", line 17,
    assert False, "I failed"
AssertionError: I failed
</failure>
```

```
  </testcase>
  <testcase classname="pkg1.test.test_things.SomeTests" name="test_ok" time="0.000093" />
  <testcase classname="pkg1.test.test_things.SomeTests" name="test_params_method:1" time="0.000099" />
  <testcase classname="pkg1.test.test_things.SomeTests" name="test_params_method:2" time="0.000101">
    <failure message="test failure">Traceback (most recent call last):
  File "nose2/plugins/loader/parameters.py", line 144, in _method
    return method(self, *argSet)
  File "nose2/tests/functional/support/scenario/tests_in_package/pkg1/test/test_things.py", line 29,
    self.assertEqual(a, 1)
AssertionError: 2 != 1
</failure>
  </testcase>
  <testcase classname="pkg1.test.test_things.SomeTests" name="test_skippy" time="0.000104">
    <skipped />
  </testcase>
  <testcase classname="pkg1.test.test_things.SomeTests" name="test_typeerr" time="0.000096">
    <error message="test failure">Traceback (most recent call last):
  File "nose2/tests/functional/support/scenario/tests_in_package/pkg1/test/test_things.py", line 13,
    raise TypeError("oops")
TypeError: oops
</error>
  </testcase>
  <testcase classname="pkg1.test.test_things.SomeTests" name="test_gen_method:1" time="0.000094" />
  <testcase classname="pkg1.test.test_things.SomeTests" name="test_gen_method:2" time="0.000090">
    <failure message="test failure">Traceback (most recent call last):
  File "nose2/plugins/loader/generators.py", line 145, in method
    return func(*args)
  File "nose2/tests/functional/support/scenario/tests_in_package/pkg1/test/test_things.py", line 24,
    assert x == 1
AssertionError
</failure>
  </testcase>
</testsuite>
```

Selecting tests with attributes

Note: New in version 0.2

Filter tests by attribute, excluding any tests whose attributes do not match any of the specified attributes.

Attributes may be simple values or lists, and may be attributes of a test method (or function), a test case class, or the callable yielded by a generator test.

Given the following test module, the attrib plugin can be used to select tests in the following ways (and others!):

Note: All examples assume the attrib plugin has been activated in a config file:

```
[unittest]
plugins = nose2.plugins.attrib
```

```
import unittest


class Test(unittest.TestCase):
```

```
    def test_fast(self):
        pass
    test_fast.fast = 1
    test_fast.layer = 2
    test_fast.flags = ['blue', 'green']

    def test_faster(self):
        pass
    test_faster.fast = 1
    test_faster.layer = 1
    test_faster.flags = ['red', 'green']

    def test_slow(self):
        pass
    test_slow.fast = 0
    test_slow.slow = 1
    test_slow.layer = 2

    def test_slower(self):
        pass
    test_slower.slow = 1
    test_slower.layer = 3
    test_slower.flags = ['blue', 'red']
```

Select tests having an attribute

Running nose2 like this:

```
nose2 -v -A fast
```

Runs these tests:

```
test_fast (attrib_example.Test) ... ok
test_faster (attrib_example.Test) ... ok
```

This selects all tests that define the attribute as any True value.

Select tests that do not have an attribute

Running nose2 like this:

```
nose2 -v -A '!fast'
```

Runs these tests:

```
test_slow (attrib_example.Test) ... ok
test_slower (attrib_example.Test) ... ok
```

This selects all tests that define the attribute as a False value, *and those tests that do not have the attribute at all*.

Select tests having an attribute with a particular value Running nose2 like this:

```
nose2 -v -A layer=2
```

Runs these tests:

```
test_fast (attrib_example.Test) ... ok
test_slow (attrib_example.Test) ... ok
```

This selects all tests that define the attribute with a matching value. The attribute value of each test case is converted to a string before comparison with the specified value. Comparison is case-insensitive.

Select tests having a value in a list attribute Running nose2 like this:

```
nose2 -v -A flags=red
```

Runs these tests:

```
test_faster (attrib_example.Test) ... ok
test_slower (attrib_example.Test) ... ok
```

Since the `flags` attribute is a list, this test selects all tests with the value `red` in their `flags` attribute. Comparison done after string conversion and is case-insensitive.

Select tests that do not have a value in a list attribute Running nose2 like this:

```
nose2 -v -A '!flags=red'
```

Runs these tests:

```
test_fast (attrib_example.Test) ... ok
```

The result in this case can be somewhat counter-intuitive. What the attrib plugin selects when you negate an attribute that is in a list are only those tests that *have the list attribute* but *without the value* specified. Tests that do not have the attribute at all are *not* selected.

Select tests using Python expressions For more complex cases, you can use the `-E` command-line option to pass a Python expression that will be evaluated in the context of each test case. Only those test cases where the expression evaluates to True (and doesn't raise an exception) will be selected.

Running nose2 like this:

```
-nose2 -v -E '"blue" in flags and layer > 2'
```

Runs only one test:

```
test_slower (attrib_example.Test) ... ok
```

Command-line options

-A DEFAULT, **-attribute** DEFAULT
: Select tests with matching attribute

-E DEFAULT, **-eval-attribute** DEFAULT
: Select tests for whose attributes the given Python expression evalures to True

Plugin class reference: AttributeSelector

class nose2.plugins.attrib.**AttributeSelector**
: Filter tests by attribute

  handleArgs(*args*)
  : Register if any attribs defined

moduleLoadedSuite(*event*)
Filter event.suite by specified attributes

Running Tests in Parallel with Multiple Processes

Note: New in version 0.3

Use the `mp` plugin to enable distribution of tests across multiple processes. Doing his may speed up your test run if your tests are heavily IO or CPU bound. But it *imposes an overhead cost* that is not trivial, and it *complicates the use of test fixtures* and may *conflict with plugins that are not designed to work with it*.

Usage

To activate the plugin, include the plugin module in the plugins list in `[unittest]` section in a config file:

```
[unittest]
plugins = nose2.plugins.mp
```

Or pass the module with the *--plugin* command-line option:

```
nose2 --plugin=nose2.plugin.mp
```

Then configure the number of processes to run. You can do that either with the *-N* option:

```
nose2 -N 2
```

or by setting `processes` in the `[multiprocess]` section of a config file:

```
[multiprocess]
processes = 2
```

Note: If you make the plugin always active by setting `always-on` in the `[multiprocess]` section of a config file, but do not set `processes` or pass *-N*, the number of processes defaults to the number of CPUs available.

Guidelines for Test Authors

Not every test suite will work well, or work at all, when run in parallel. For some test suites, parallel execution makes no sense. For others, it will expose bugs and ordering dependencies test cases and test modules.

Overhead Cost Starting subprocesses and dispatching tests takes time. A test run that includes a relatively small number of tests that are not IO or CPU bound (or calling time.sleep()) is likely to be *slower* when run in parallel. As of this writing, for instance, nose2's test suite takes about 10 times as long to run when using multiprocessing, due to the overhead cost.

Shared Fixtures The individual test processes do not share state or data after launch. This means *tests that share a fixture* – tests that are loaded from modules where `setUpModule` is defined, and tests in test classes that define `setUpClass` – *must all be dispatched to the same process at the same time*. So if you use these kinds of fixtures, your test runs may be less parallel than you expect.

Tests Load Twice Test cases may not be pickleable, so nose2 can't transmit them directly to its test runner processes. Tests are distributed by name. This means that *tests always load twice* – once in the main process, during initial collection, and then again in the test runner process, where they are loaded by name. This may be problematic for some test suites.

Random Execution Order Tests do not execute in the same order when run in parallel. Results will be returned in effectively random order, and tests in the same module (*as long as they do not share fixtures*) may execute in any order and in different processes. Some tests suites have ordering dependencies, intentional or not, and those that do will fail randomly when run with this plugin.

Guidelines for Plugin Authors

The MultiProcess plugin is designed to work with other plugins. But other plugins may have to return the favor, especially if they load tests or care about something that happens *during* test execution.

New Methods The MultiProcess plugin adds a few plugin hooks that other plugins can use to set themselves up for multiprocess test runs. Plugins don't have to do anything special to register for these hooks, just implement the methods as normal.

registerInSubprocess(*self*, *event*)

> **Parameters** **event** – nose2.plugins.mp.RegisterInSubprocessEvent
>
> The `registerInSubprocess` hook is called after plugin registration to enable plugins that need to run in subprocesses to register that fact. The most common thing to do, for plugins that need to run in subprocesses, is:
>
> ```
> def registerInSubprocess(self, event):
>     event.pluginClasses.append(self.__class__)
> ```
>
> It is not required that plugins append their own class. If for some reason there is a different plugin class, or set of classes, that should run in the test-running subprocesses, add that class or those classes instead.

startSubprocess(*self*, *event*)

> **Parameters** **event** – nose2.plugins.mp.SubprocessEvent
>
> The `startSubprocess` hook fires in each test-running subprocess after it has loaded its plugins but before any tests are executed.
>
> Plugins can customize test execution here in the same way as in `startTestRun()`, by setting `event.executeTests`, and prevent test execution by setting `event.handled` to True and returning False.

stopSubprocess(*self*, *event*)

> **Parameters** **event** – nose2.plugins.mp.SubprocessEvent
>
> The `stopSubprocess` event fires just before each test running subprocess shuts down. Plugins can use this hook for any per-process finalization that they may need to do.
>
> The same event instance is passed to `startSubprocess` and `stopSubprocess`, which enables plugins to use that event's metadata to communicate state or other information from the start to the stop hooks, if needed.

New Events The MultiProcess plugin's new hooks come with custom event classes.

class nose2.plugins.mp.**RegisterInSubprocessEvent**(***metadata*)
Event fired to notify plugins that multiprocess testing will occur

pluginClasses
Add a plugin class to this list to cause the plugin to be instantiated in each test-running subprocess. The most common thing to do, for plugins that need to run in subprocesses, is:

```
def registerInSubprocess(self, event):
    event.pluginClasses.append(self.__class__)
```

class nose2.plugins.mp.**SubprocessEvent**(*loader*, *result*, *runner*, *plugins*, *connection*, ***metadata*)
Event fired at start and end of subprocess execution.

loader
Test loader instance

result
Test result

plugins
List of plugins loaded in the subprocess.

connection
The multiprocessing.Connection instance that the subprocess uses for communication with the main process.

executeTests
Callable that will be used to execute tests. Plugins may set this attribute to wrap or otherwise change test execution. The callable must match the signature:

```
def execute(suite, result):
    ...
```

Stern Warning All event attributes, *including ''event.metadata'', must be pickleable*. If your plugin sets any event attributes or puts anything into event.metadata, it is your responsibility to ensure that anything you can possibly put in is pickleable.

Do I Really Care? If you answer *yes* to any of the following questions, then your plugin will not work with multiprocess testing without modification:

- Does your plugin load tests?
- Does your plugin capture something that happens during test execution?
- Does your plugin require user interaction during test execution?
- Does your plugin set executeTests in startTestRun?

Here's how to handle each of those cases.

Loading Tests

- Implement registerInSubprocess() as suggested to enable your plugin in the test runner processes.

Capturing Test Execution State

- Implement `registerInSubprocess()` as suggested to enable your plugin in the test runner processes.
- Be wary of setting `event.metadata` unconditionally. Your plugin will execute in the main process and in the test runner processes, and will see `setTestOutcome()` and `testOutcome()` events *in both processes*. If you unconditionally set a key in `event.metadata`, the plugin instance in the main process will overwrite anything set in that key by the instance in the subprocess.
- If you need to write something to a file, implement `stopSubprocess()` to write a file in each test runner process.

Overriding Test Execution

- Implement `registerInSubprocess()` as suggested to enable your plugin in the test runner processes and make a note that your plugin is running under a multiprocess session.
- When running multiprocess, *do not* set `event.executeTests` in `startTestRun()` – instead, set it in `startSubprocess()` instead. This will allow the multiprocess plugin to install its test executor in the main process, while your plugin takes over test execution in the test runner subprocesses.

Interacting with Users

- You are probably safe because as a responsible plugin author you are already firing the interaction hooks (`beforeInteraction()`, `afterInteraction()`) around your interactive bits, and skipping them when the `beforeInteraction()` hook returns false and sets `event.handled`.

  If you're not doing that, start!

Reference

Configuration [multiprocess]

always-on

Default False

Type boolean

processes

Default 2

Type integer

test-run-timeout

Default 60.0

Type float

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[multiprocess]
always-on = False
processes = 2
test-run-timeout = 60.0
```

Command-line options
-N DEFAULT, **-processes** DEFAULT
    # o procs

Plugin class reference: MultiProcess
class nose2.plugins.mp.**MultiProcess**

Organizing Test Fixtures into Layers

Note: New in version 0.4

Layers allow more flexible organization of test fixtures than test-, class- and module- level fixtures. Layers in nose2 are inspired by and aim to be compatible with the layers used by Zope's testrunner.

Using layers, you can do things like:

- Implement package-level fixtures by sharing a layer among all test cases in the package.
- Share fixtures across tests in different modules without having them run multiple times.
- Create a fixture tree deeper than three levels (test, class and module).
- Make fixtures available for other packages or projects to use.

A layer is a *new-style* class that implements at least a `setUp` classmethod:

```
class Layer(object):
    @classmethod
    def setUp(cls):
        # ...
```

It may also implement `tearDown`, `testSetUp` and `testTearDown`, all as classmethods.

To assign a layer to a test case, set the test case's `layer` property:

```
class Test(unittest.TestCase):
    layer = Layer
```

Note that the layer *class* is assigned, not an instance of the layer. Typically layer classes are not instantiated.

Sub-layers

Layers may subclass other layers:

```
class SubLayer(Layer):
    @classmethod
    def setUp(cls):
        # ...
```

In this case, all tests that belong to the sub-layer also belong to the base layer. For example for this test case:

```
class SubTest(unittest.TestCase):
    layer = SubLayer
```

The `setUp` methods from *both* `SubLayer` and `Layer` will run before any tests are run. The superclass's setup will always run before the subclass's setup. For teardown, the reverse: the subclass's teardown runs before the superclass's.

Warning: One important thing to note: layers that subclass other layers *must not* call their superclass's `setUp`, `tearDown`, etc. – the test runner will take care of organizing tests so that the superclass's methods are called in the right order:

```
Layer.setUp ->
  SubLayer.setUp ->
    Layer.testSetUp ->
      SubLayer.testSetUp ->
        TestCase.setUp
          TestCase.run
        TestCase.tearDown
      SubLayer.testTearDown <-
    Layer.testTearDown <-
  SubLayer.tearDown <-
Layer.tearDown <-
```

If a sublayer calls it superclass's methods directly, *those methods will be called twice*.

Layer method reference

class `Layer`

Not an actual class, but reference documentation for the methods layers can implement. There is no layer base class. Layers must be subclasses of `object` or other layers.

classmethod `setUp` (*cls*)

The layer's `setUp` method is called before any tests belonging to that layer are executed. If no tests belong to the layer (or one of its sub-layers) then the `setUp` method will not be called.

classmethod `tearDown` (*cls*)

The layer's `tearDown` method is called after any tests belonging to the layer are executed, if the layer's `setUp` method was called and did not raise an exception. It will not be called if the layer has no `setUp` method, or if that method did not run or did raise an exception.

classmethod `testSetUp` (*cls*[, *test*])

The layer's `testSetUp` method is called before each test belonging to the layer (and its sub-layers). If the method is defined to accept an argument, the test case instance is passed to the method. The method may also be defined to take no arguments.

classmethod `testTearDown` (*cls*[, *test*])

The layer's `testTearDown` method is called after each test belonging to the layer (and its sub-layers), if the layer also defines a `setUpTest` method and that method ran successfully (did not raise an exception) for this test case.

Layers DSL

nose2 includes a DSL for setting up layer-using tests called "such". Read all about it here: *Such: a Functional-Test Friendly DSL*.

Pretty reports

The layers plugin module includes a second plugin that alters test report output to make the layer groupings more clear. When activated with the *`--layer-reporter`* command-line option (or via a config file), test output that normally looks like this:

```
test (test_layers.NoLayer) ... ok
test (test_layers.Outer) ... ok
test (test_layers.InnerD) ... ok
test (test_layers.InnerA) ... ok
test (test_layers.InnerA_1) ... ok
test (test_layers.InnerB_1) ... ok
test (test_layers.InnerC) ... ok
test2 (test_layers.InnerC) ... ok

----------------------------------------------------------------------
Ran 8 tests in 0.001s

OK
```

Will instead look like this:

```
test (test_layers.NoLayer) ... ok
Base
  test (test_layers.Outer) ... ok
  LayerD
    test (test_layers.InnerD) ... ok
  LayerA
    test (test_layers.InnerA) ... ok
  LayerB
    LayerC
      test (test_layers.InnerC) ... ok
      test2 (test_layers.InnerC) ... ok
    LayerB_1
      test (test_layers.InnerB_1) ... ok
    LayerA_1
      test (test_layers.InnerA_1) ... ok

----------------------------------------------------------------------
Ran 8 tests in 0.002s

OK
```

The layer reporter plugin can also optionally colorize the keywords ('A', 'having', and 'should' by default) in output from tests defined with the *such DSL*.

If you would like to change how the layer is displayed you need to set the description attribute.

```
class LayerD(Layer):
    description = '*** This is a very important custom layer description ***'
```

Now the output will be the following:

```
test (test_layers.NoLayer) ... ok
Base
  test (test_layers.Outer) ... ok
  *** This is a very important custom layer description ***
    test (test_layers.InnerD) ... ok
  LayerA
    test (test_layers.InnerA) ... ok
  LayerB
    LayerC
      test (test_layers.InnerC) ... ok
      test2 (test_layers.InnerC) ... ok
    LayerB_1
      test (test_layers.InnerB_1) ... ok
```

```
    LayerA_1
      test (test_layers.InnerA_1) ... ok

----------------------------------------------------------------------
Ran 8 tests in 0.002s

OK
```

Warnings and Caveats

Test case order and module isolation Test cases that use layers will not execute in the same order as test cases that do not. In order to execute the layers efficiently, the test runner must reorganize *all* tests in the loaded test suite to group those having like layers together (and sub-layers under their parents). If you share layers across modules this may result in tests from one module executing interleaved with tests from a different module.

Mixing layers with setUpClass and module fixtures Don't cross the streams.

The implementation of class- and module-level fixtures in unittest2 depends on introspecting the class hierarchy inside of the unittest.TestSuite. Since the suites that the layers plugin uses to organize tests derive from `unittest.BaseTestSuite` not `unittest.TestSuite`, class- and module- level fixtures in TestCase classes that use layers will be ignored.

Mixing layers and multiprocess testing In the initial release, *test suites using layers are incompatible with the multiprocess plugin*. This should be fixed in a future release.

Plugin reference

Configuration [layer-reporter]

`always-on`

Default False

Type boolean

`colors`

Default False

Type boolean

`highlight-words`

Default ['A', 'having', 'should']

Type list

`indent`

Default

Type str

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[layer-reporter]
always-on = False
colors = False
highlight-words = A
                  having
                  should
indent =
```

Command-line options

-layer-reporter DEFAULT
Add layer information to test reports

Plugin class reference: LayerReporter
class nose2.plugins.layers.**LayerReporter**

Plugin class reference: Layers
class nose2.plugins.layers.**Layers**

Loader: Doctests

Load tests from doctests.

This plugin implements `handleFile()` to load doctests from text files and python modules.

To disable loading doctests from text files, configure an empty extensions list:

```
[doctest]
extensions =
```

Configuration [doctest]

always-on

Default False

Type boolean

extensions

Default ['.txt', '.rst']

Type list

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[doctest]
always-on = False
extensions = .txt
             .rst
```

Command-line options

-with-doctest DEFAULT
Load doctests from text files and modules

Plugin class reference: DocTestLoader

class nose2.plugins.doctests.**DocTestLoader**

handleFile(*event*)
Load doctests from text files and modules

Mapping exceptions to test outcomes

Map exceptions to test outcomes.

This plugin implements `setTestOutcome()` to enable simple mapping of exception classes to existing test outcomes.

By setting a list of exception classes in a nose2 config file, you can configure exceptions that would otherwise be treated as test errors, to be treated as failures or skips instead:

```
[outcomes]
always-on = True
treat-as-fail = NotImplementedError
treat-as-skip = TodoError
                IOError
```

Configuration [outcomes]

always-on

Default False

Type boolean

treat-as-fail

Default []

Type list

treat-as-skip

Default []

Type list

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[outcomes]
always-on = False
treat-as-fail =
treat-as-skip =
```

Command-line options

-set-outcomes DEFAULT
Treat some configured exceptions as failure or skips

Plugin class reference: Outcomes

class nose2.plugins.outcomes.**Outcomes**
Map exceptions to other test outcomes

setTestOutcome(*event*)
Update outcome, exc_info and reason based on configured mappings

Collecting tests without running them

This plugin implements `startTestRun()`, setting a test executor (`event.executeTests`) that just collects tests without executing them. To do so it calls result.startTest, result.addSuccess and result.stopTest for ech test, without calling the test itself.

Configuration [collect-only]

always-on

Default False

Type boolean

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[collect-only]
always-on = False
```

Command-line options

-collect-only DEFAULT
Collect and output test names, do not run any tests

Plugin class reference: CollectOnly

class nose2.plugins.collect.**CollectOnly**
Collect but don't run tests

collectTests(*suite*, *result*)
Collect tests but don't run them

startTestRun(*event*)
Replace event.executeTests

Using Test IDs

Allow easy test selection with test ids.

Assigns (and, in verbose mode, prints) a sequential test id for each test executed. Ids can be fed back in as test names, and this plugin will translate them back to full test names. Saves typing!

This plugin implements `reportStartTest()`, `loadTestsFromName()`, `loadTestsFromNames()` and `stopTest()`.

Configuration [testid]

always-on

Default False

Type boolean

id-file

Default .noseids

Type str

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[testid]
always-on = False
id-file = .noseids
```

Command-line options

-I DEFAULT, **-with-id** DEFAULT
Add test ids to output

Plugin class reference: TestId

class nose2.plugins.testid.**TestId**
Allow easy test select with ids

loadIds()
Load previously pickled 'ids' and 'tests' attributes.

loadTestsFromName(*event*)
Load tests from a name that is an id

If the name is a number, it might be an ID assigned by us. If we can find a test to which we have assigned that ID, event.name is changed to the test's real ID. In this way, tests can be referred to via sequential numbers.

loadTestsFromNames(*event*)
Translate test ids into test names

nextId()
Increment ID and return it.

reportStartTest(*event*)
Record and possibly output test id

stopTestRun(*event*)
Write testids file

Profiling

Profile test execution using hotshot.

This plugin implements `startTestRun()` and replaces `event.executeTests` with `hotshot.Profile.runcall()`. It implements `beforeSummaryReport()` to output profiling information before the final test summary time. Config file options `filename`, `sort` and `restrict` can be used to change where profiling information is saved and how it is presented.

Configuration [profiler]

always-on

Default False

Type boolean

filename

Default

Type str

restrict

Default []

Type list

sort

Default cumulative

Type str

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[profiler]
always-on = False
filename =
restrict =
sort = cumulative
```

Command-line options

-P DEFAULT, **-profile** DEFAULT
Run tests under profiler

Plugin class reference: Profiler

class nose2.plugins.prof.**Profiler**
Profile the test run

beforeSummaryReport(*event*)
Output profiling results

register()
Don't register if hotshot is not found

startTestRun(*event*)
Set up the profiler

Tracing hook execution

This plugin is primarily useful for plugin authors who want to debug their plugins.

It prints each hook that is called to stderr, along with details of the event that was passed to the hook.

To do that, this plugin overrides `nose2.events.Plugin.register()` and, after registration, replaces all existing `nose2.events.Hook` instances in `session.hooks` with instances of a Hook subclass that prints information about each call.

Configuration [print-hooks]

always-on

Default False

Type boolean

Sample configuration The default configuration is equivalent to including the following in a unittest.cfg file.

```
[print-hooks]
always-on = False
```

Command-line options

-print-hooks DEFAULT
Print names of hooks in order of execution

Plugin class reference: PrintHooks

class nose2.plugins.printhooks.**PrintHooks**
Print hooks as they are called

register()
Override to inject noisy hook instances.

Replaces Hook instances in `self.session.hooks.hooks` with noisier objects.

Sample output

PrintHooks output for a test run that discovers one standard TestCase test in a python module.

Hooks that appear indented are called from within other hooks.

```
handleArgs: CommandLineArgsEvent(handled=False, args=Namespace(collect_only=None, config=['unittest.c

createTests: CreateTestsEvent(loader=<PluggableTestLoader>, testNames=[], module=<module '__main__' f

loadTestsFromNames: LoadFromNames(names=[], module=None)

  handleFile: HandleFileEvent(handled=False, loader=<PluggableTestLoader>, name='tests.py', path='nos

  matchPath: MatchPathEvent(handled=False, name='tests.py', path='nose2/tests/functional/support/scen
```

```
  loadTestsFromModule: LoadFromModuleEvent(handled=False, loader=<PluggableTestLoader>, module=<modul
    loadTestsFromTestCase: LoadFromTestCaseEvent(handled=False, loader=<PluggableTestLoader>, testCas
    getTestCaseNames: GetTestCaseNamesEvent(handled=False, loader=<PluggableTestLoader>, testCase=<cl
  handleFile: HandleFileEvent(handled=False, loader=<PluggableTestLoader>, name='tests.pyc', path='no
runnerCreated: RunnerCreatedEvent(handled=False, runner=<PluggableTestRunner>)
resultCreated: ResultCreatedEvent(handled=False, result=<PluggableTestResult>)
startTestRun: StartTestRunEvent(handled=False, runner=<PluggableTestRunner>, suite=<unittest2.suite.T
startTest: StartTestEvent(handled=False, test=<tests.Test testMethod=test>, result=<PluggableTestResu
  reportStartTest: ReportTestEvent(handled=False, testEvent=<nose2.events.StartTestEvent object at 0x
setTestOutcome: TestOutcomeEvent(handled=False, test=<tests.Test testMethod=test>, result=<Pluggable
testOutcome: TestOutcomeEvent(handled=False, test=<tests.Test testMethod=test>, result=<PluggableTest
  reportSuccess: ReportTestEvent(handled=False, testEvent=<nose2.events.TestOutcomeEvent object at 0x
.
stopTest: StopTestEvent(handled=False, test=<tests.Test testMethod=test>, result=<PluggableTestResult
stopTestRun: StopTestRunEvent(handled=False, runner=<PluggableTestRunner>, result=<PluggableTestResul
afterTestRun: StopTestRunEvent(handled=False, runner=<PluggableTestRunner>, result=<PluggableTestResu

  beforeErrorList: ReportSummaryEvent(handled=False, stopTestEvent=<nose2.events.StopTestRunEvent obj
----------------------------------------------------------------------
  beforeSummaryReport: ReportSummaryEvent(handled=False, stopTestEvent=<nose2.events.StopTestRunEvent
Ran 1 test in 0.001s

  wasSuccessful: ResultSuccessEvent(handled=False, result=<PluggableTestResult>, success=False)
OK
  afterSummaryReport: ReportSummaryEvent(handled=False, stopTestEvent=<nose2.events.StopTestRunEvent
```

1.5.3 Third-party Plugins

If you are a plugin author, please add your plugin to the list on the nose2 wiki. If you are looking for more plugins, check that list!

1.6 Tools and Helpers

1.6.1 Tools for Test Authors

Parameterized tests

nose2.tools.**params**(**paramList*)

Make a test function or method parameterized.

```
import unittest

from nose2.tools import params


@params(1, 2, 3)
def test_nums(num):
    assert num < 4


class Test(unittest.TestCase):

    @params((1, 2), (2, 3), (4, 5))
    def test_less_than(self, a, b):
        assert a < b
```

Parameters in the list may be defined as simple values, or as tuples. To pass a tuple as a simple value, wrap it in another tuple.

See also: *Loader: Parameterized Tests*

Such: a Functional-Test Friendly DSL

Note: New in version 0.4

Such is a DSL for writing tests with expensive, nested fixtures – which typically means functional tests. It requires the layers plugin (see *Organizing Test Fixtures into Layers*).

What does it look like?

Unlike some python testing DSLs, such is just plain old python.

```
import unittest

from nose2.tools import such


class SomeLayer(object):

    @classmethod
    def setUp(cls):
        it.somelayer = True

    @classmethod
    def tearDown(cls):
```

```
        del it.somelayer

#
# Such tests start with a declaration about the system under test
# and will typically bind the test declaration to a variable with
# a name that makes nice sentences, like 'this' or 'it'.
#
with such.A('system with complex setup') as it:

    #
    # Each layer of tests can define setup and teardown methods.
    # setup and teardown methods defined here run around the entire
    # group of tests, not each individual test.
    #
    @it.has_setup
    def setup():
        it.things = [1]

    @it.has_teardown
    def teardown():
        it.things = []

    #
    # The 'should' decorator is used to mark tests.
    #
    @it.should('do something')
    def test():
        assert it.things
        #
        # Tests can use all of the normal unittest TestCase assert
        # methods by calling them on the test declaration.
        #
        it.assertEqual(len(it.things), 1)

    #
    # The 'having' context manager is used to introduce a new layer,
    # one that depends on the layer(s) above it. Tests in this
    # new layer inherit all of the fixtures of the layer above.
    #
    with it.having('an expensive fixture'):
        @it.has_setup
        def setup():
            it.things.append(2)

        #
        # Tests that take an argument will be passed the
        # unittest.TestCase instance that is generated to wrap
        # them. Tests can call any and all TestCase methods on this
        # instance.
        #
        @it.should('do more things')
        def test(case):
            case.assertEqual(it.things[-1], 2)

        #
        # Layers can be nested to any depth.
        #
        with it.having('another precondtion'):
```

```
    @it.has_setup
    def setup():
        it.things.append(3)

    @it.has_teardown
    def teardown():
        it.things.pop()

    @it.should('do that not this')
    def test(case):
        it.things.append(4)
        #
        # Tests can add their own cleanup functions.
        #
        case.addCleanup(it.things.pop)
        case.assertEqual(it.things[-1], 4, it.things)

    @it.should('do this not that')
    def test(case):
        case.assertEqual(it.things[-1], 3, it.things[:])

#
# A layer may have any number of sub-layers.
#
with it.having('a different precondition'):

    #
    # A layer defined with ``having`` can make use of
    # layers defined elsewhere. An external layer
    # pulled in with ``it.uses`` becomes a parent
    # of the current layer (though it doesn't actually
    # get injected into the layer's MRO).
    #
    it.uses(SomeLayer)

    @it.has_setup
    def setup():
        it.things.append(99)

    @it.has_teardown
    def teardown():
        it.things.pop()

    #
    # Layers can define setup and teardown methods that wrap
    # each test case, as well, corresponding to TestCase.setUp
    # and TestCase.tearDown.
    #
    @it.has_test_setup
    def test_setup(case):
        it.is_funny = True
        case.is_funny = True

    @it.has_test_teardown
    def test_teardown(case):
        delattr(it, 'is_funny')
        delattr(case, 'is_funny')
```

```
            @it.should('do something else')
            def test(case):
                assert it.things[-1] == 99
                assert it.is_funny
                assert case.is_funny

            @it.should('have another test')
            def test(case):
                assert it.is_funny
                assert case.is_funny

            @it.should('have access to an external fixture')
            def test(case):
                assert it.somelayer

            with it.having('a case inside the external fixture'):
                @it.should('still have access to that fixture')
                def test(case):
                    assert it.somelayer

#
# To convert the layer definitions into test cases, you have to call
# 'createTests' and pass in the module globals, so that the test cases
# and layer objects can be inserted into the module.
#
it.createTests(globals())


#
# Such tests and normal tests can coexist in the same modules.
#
class NormalTest(unittest.TestCase):

    def test(self):
        pass
```

The tests it defines are unittest tests, and can be used with nose2 with just the layers plugin. You also have the option of activating a reporting plugin (`nose2.plugins.layers.LayerReporter`) to provide a more discursive brand of output:

```
test (test_such.NormalTest) ... ok
A system with complex setup
  should do something ... ok
  having an expensive fixture
    should do more things ... ok
    having another precondtion
      should do that not this ... ok
      should do this not that ... ok
    having a different precondition
      should do something else ... ok
      should have another test ... ok

----------------------------------------------------------------------
Ran 7 tests in 0.002s

OK
```

How does it work?

Such uses the things in python that are most like anonymous code blocks to allow you to construct tests with meaningful names and deeply-nested fixtures. Compared to DSLs in languages that do allow blocks, it is a little bit more verbose – the block-like decorators that mark fixture methods and test cases need to decorate *something*, so each fixture and test case has to have a function definition. You can use the same function name over and over here, or give each function a meaningful name.

The set of tests begins with a description of the system under test as a whole, marked with the `A` context manager:

```
from nose2.tools import such

with such.A('system described here') as it:
    # ...
```

Groups of tests are marked by the `having` context manager:

```
with it.having('a description of a group'):
    # ...
```

Within a test group (including the top-level group), fixtures are marked with decorators:

```
@it.has_setup
def setup():
    # ...

@it.has_test_setup
def setup_each_test_case():
    # ...
```

And tests are likewise marked with the `should` decorator:

```
@it.should('exhibit the behavior described here')
def test(case):
    # ...
```

Test cases may optionally take one argument. If they do, they will be passed the `unittest.TestCase` instance generated for the test. They can use this TestCase instance to execute assert methods, among other things. Test functions can also call assert methods on the top-level scenario instance, if they don't take the `case` argument:

```
@it.should("be able to use the scenario's assert methods")
def test():
    it.assertEqual(something, 'a value')

@it.should("optionally take an argument")
def test(case):
    case.assertEqual(case.attribute, 'some value')
```

Finally, to actually generate tests, you **must** call `createTests` on the top-level scenario instance:

```
it.createTests(globals())
```

This call generates the `unittest.TestCase` instances for all of the tests, and the layer classes that hold the fixtures defined in the test groups. See *Organizing Test Fixtures into Layers* for more about test layers.

Running tests Since order is often significant in functional tests, **such DSL tests always execute in the order in which they are defined in the module**. Parent groups run before child groups, and sibling groups and sibling tests within a group execute in the order in which they are defined.

Otherwise, tests written in the such DSL are collected and run just like any other tests, with one exception: their names. The name of a such test case is the name of its immediately surrounding group, plus the description of the test, prepended with `test ####:`, where '####' is the test's (0-indexed) position within its group. To run a case individually, you must pass in this full name – usually you'll have to quote it. For example, to run the case `should do more things` defined above (assuming the layers plugin is activated by a config file, and the test module is in the normal path of test collection), you would run nose2 like this:

```
nose2 "test_such.having an expensive fixture.test 0000: should do more things"
```

That is, for the a generated test case, the **group description** is the **class name**, and the **test case description** is the **test case name**. As you can see if you run an individual test with the layer reporter active, all of the group fixtures execute in proper order when a test is run individually:

```
$ nose2 "test_such.having an expensive fixture.test 0000: should do more things"
A system with complex setup
  having an expensive fixture
    should do more things ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Reference

nose2.tools.such.**A**(**args*, ***kwds*)

Test scenario context manager.

Returns a `nose2.tools.such.Scenario` instance, which by convention is bound to `it`:

```
with such.A('test scenario') as it:
    # tests and fixtures
```

class nose2.tools.such.**Scenario**(*description*)

A test scenario.

A test scenario defines a set of fixtures and tests that depend on those fixtures.

createTests(*mod*)

Generate test cases for this scenario.

> **Warning:** You must call this, passing in `globals()`, to generate tests from the scenario. If you don't call `createTests`, **no tests will be created**.

```
it.createTests(globals())
```

has_setup(*func*)

Add a setup method to this group.

The setup method will run once, before any of the tests in the containing group.

A group may define any number of setup functions. They will execute in the order in which they are defined.

```
@it.has_setup
def setup():
    # ...
```

has_teardown (*func*)
Add a teardown method to this group.

The teardown method will run once, after all of the tests in the containing group.

A group may define any number of teardown functions. They will execute in the order in which they are defined.

```
@it.has_teardown
def teardown():
    # ...
```

has_test_setup (*func*)
Add a test case setup method to this group.

The setup method will run before each of the tests in the containing group.

A group may define any number of test case setup functions. They will execute in the order in which they are defined.

Test setup functions may optionally take one argument. If they do, they will be passed the `unittest.TestCase` instance generated for the test.

```
@it.has_test_setup
def setup(case):
    # ...
```

has_test_teardown (*func*)
Add a test case teardown method to this group.

The teardown method will run before each of the tests in the containing group.

A group may define any number of test case teardown functions. They will execute in the order in which they are defined.

Test teardown functions may optionally take one argument. If they do, they will be passed the `unittest.TestCase` instance generated for the test.

```
@it.has_test_teardown
def teardown(case):
    # ...
```

having (**args*, ***kwds*)
Define a new group under the current group.

Fixtures and tests defined within the block will belong to the new group.

```
with it.having('a description of this group'):
    # ...
```

should (*desc*)
Define a test case.

Each function marked with this decorator becomes a test case in the current group.

The decorator takes one optional argument, the description of the test case: what it **should** do. If this argument is not provided, the docstring of the decorated function will be used as the test case description.

Test functions may optionally take one argument. If they do, they will be passed the `unittest.TestCase` instance generated for the test. They can use this TestCase instance to execute assert methods, among other things.

```
@it.should('do this')
def dothis(case):
    # ....

@it.should
def dothat():
    "do that also"
    # ....
```

1.7 Changelog

1.7.1 0.4.7

- Feature: Added start-dir config option. Thanks to Stéphane Klein.
- Bug: Fixed broken import in collector.py. Thanks to Shaun Crampton.
- Bug: Fixed processes command line option in mp plugin. Thanks to Tim Sampson.
- Bug: Fixed handling of class fixtures in multiprocess plugin. Thanks to Tim Sampson.
- Bug: Fixed intermittent test failure caused by nondeterministic key ordering. Thanks to Stéphane Klein.
- Bug: Fixed syntax error in printhooks. Thanks to Tim Sampson.
- Docs: Fixed formatting in changelog. Thanks to Omer Katz.
- Docs: Added help text for verbose flag. Thanks to Tim Sampson.
- Docs: Fixed typos in docs and examples. Thanks to Tim Sampson.
- Docs: Added badges to README. Thanks to Omer Katz.
- Updated six version requirement to be less Restrictive. Thanks to Stéphane Klein.
- Cleaned up numerous PEP8 violations. Thanks to Omer Katz.

1.7.2 0.4.6

- Bug: fixed DeprecationWarning for compiler package on python 2.7. Thanks Max Arnold.
- Bug: fixed lack of timing information in junitxml exception reports. Thanks Viacheslav Dukalskiy.
- Bug: cleaned up junitxml xml output. Thanks Philip Thiem.
- Docs: noted support for python 3.3. Thanks Omer Katz for the bug report.

1.7.3 0.4.5

- Bug: fixed broken interaction between attrib and layers plugins. They can now be used together. Thanks @fajpunk.
- Bug: fixed incorrect calling order of layer setup/teardown and test setup/test teardown methods. Thanks again @fajpunk for tests and fixes.

1.7.4 0.4.4

- Bug: fixed sort key generation for layers.

1.7.5 0.4.3

- Bug: fixed packaging for non-setuptools, pre-python 2.7. Thanks to fajpunk for the patch.

1.7.6 0.4.2

- Bug: fixed unpredictable ordering of layer tests.
- Added `uses` method to `such.Scenario` to allow use of externally-defined layers in such DSL tests.

1.7.7 0.4.1

- Fixed packaging bug.

1.7.8 0.4

- New plugin: Added nose2.plugins.layers to support Zope testing style fixture layers.
- New tool: Added nose2.tools.such, a spec-like DSL for writing tests with layers.
- New plugin: Added nose2.plugins.loader.loadtests to support the unittest2 load_tests protocol.

1.7.9 0.3

- New plugin: Added nose2.plugins.mp to support distributing test runs across multiple processes.
- New plugin: Added nose2.plugins.testclasses to support loading tests from ordinary classes that are not subclasses of unittest.TestCase.
- The default script target was changed from `nose2.main` to `nose2.discover`. The former may still be used for running a single module of tests, unittest-style. The latter ignores the `module` argument. Thanks to @dtcaciuc for the bug report (#32).
- `nose2.main.PluggableTestProgram` now accepts an `extraHooks` keyword argument, which allows attaching arbitrary objects to the hooks system.
- Bug: Fixed bug that caused Skip reason to always be set to `None`.

1.7.10 0.2

- New plugin: Added nose2.plugins.junitxml to support jUnit XML output.
- New plugin: Added nose2.plugins.attrib to support test filtering by attributes.
- New hook: Added afterTestRun hook, moved result report output calls to that hook. This prevents plugin ordering issues with the stopTestRun hook (which still exists, and fires before afterTestRun).
- Bug: Fixed bug in loading of tests by name that caused ImportErrors to be silently ignored.
- Bug: Fixed missing __unittest flag in several modules. Thanks to Wouter Overmeire for the patch.

- Bug: Fixed module fixture calls for function, generator and param tests.
- Bug: Fixed passing of command-line argument values to list options. Before this fix, lists of lists would be appended to the option target. Now, the option target list is extended with the new values. Thanks to memedough for the bug report.

1.7.11 0.1

Initial release.

CHAPTER

TWO

PLUGIN DEVELOPER'S GUIDE

2.1 Writing Plugins

nose2 supports plugins for test collection, selection, observation and reporting – among other things. There are two basic rules for plugins:

- Plugin classes must subclass `nose2.events.Plugin`.
- Plugins may implement any of the methods described in the *Hook reference*.

2.1.1 Hello World

Here's a basic plugin. It doesn't do anything besides log a message at the start of a test run.

```
import logging
import os

from nose2.events import Plugin

log = logging.getLogger('nose2.plugins.helloworld')

class HelloWorld(Plugin):
    configSection = 'helloworld'
    commandLineSwitch = (None, 'hello-world', 'Say hello!')

    def startTestRun(self, event):
        log.info('Hello pluginized world!')
```

To see this plugin in action, save it into an importable module, then add that module to the `plugins` key in the `[unittest]` section of a config file loaded by nose2, such as `unittest.cfg`. Then run nose2:

```
nose2 --log-level=INFO --hello-world
```

And you should see the log message before the first dot appears.

2.1.2 Loading plugins

As mentioned above, for nose2 to find a plugin, it must be in an importable module, and the module must be listed under the `plugins` key in the `[unittest]` section of a config file loaded by nose2:

```
[unittest]
plugins = mypackage.someplugin
          otherpackage.thatplugin
          thirdpackage.plugins.metoo
```

As you can see, plugin *modules* are listed, one per line. All plugin classes in those modules will be loaded – but not necessarily active. Typically plugins do not activate themselves ("register") without seeing a command-line flag, or `always-on = True` in their config file section.

2.1.3 Command-line Options

nose2 uses argparse for command-line argument parsing. Plugins may enable command-line options that register them as active, or take arguments or flags controlling their operation.

The most basic thing to do is to set the plugin's `commandLineSwitch` attribute, which will automatically add a command-line flag that registers the plugin.

To add other flags or arguments, you can use the Plugin methods `nose2.events.Plugin.addFlag()`, `nose2.events.Plugin.addArgument()` or `nose2.events.Plugin.addOption()`. If those don't offer enough flexibility, you can directly manipulate the argument parser by accessing `self.session.argparse` or the plugin option group by accessing `self.session.pluginargs`.

Please note though that the *majority* of your plugin's configuration should be done via config file options, not command line options.

2.1.4 Config File Options

Plugins may specify a config file section that holds their configuration by setting their `configSection` attribute. All plugins, regardless of whether they specify a config section, have a `config` attribute that holds a `nose2.config.Config` instance. This will be empty of values if the plugin does not specify a config section or if no loaded config file includes that section.

Plugins should extract the user's configuration selections from their config attribute in their `__init__` methods. Plugins that want to use nose2's Sphinx extension to automatically document themselves **must** do so.

Config file options may be extracted as strings, ints, booleans or lists.

You should provide reasonable defaults for all config options.

2.1.5 Guidelines

Events

nose2's plugin api is based on the api in unittest2's under-development plugins branch. It differs from nose's plugins api in one major area: what it passes to hooks. Where nose passes a variety of arguments, nose2 *always passes an event*. The events are listed in the *Event reference*.

Here's the key thing about that: *event attributes are read-write*. Unless stated otherwise in the documentation for a hook, you can set a new value for any event attribute, and *this will do something*. Plugins and nose2 systems will see that new value and either use it instead of what was originally set in the event (example: the reporting stream or test executor), or use it to supplement something they find elsewhere (example: extraTests on a test loading event).

"Handling" events

Many hooks give plugins a chance to completely handle events, bypassing other plugins and any core nose2 operations. To do this, a plugin sets `event.handled` to True and, generally, returns an appropriate value from the hook method. What is an appropriate value varies by hook, and some hooks *can't* be handled in this way. But even for hooks where handling the event doesn't stop all processing, it *will* stop subsequently-loaded plugins from seeing the event.

Logging

nose2 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 `nose2.plugins` namespace.

2.1.6 Recipes

- Writing a plugin that monitors or controls test result output

  Implement any of the `report*` hook methods, especially if you want to output to the console. If outputing to file or other system, you might implement `testOutcome()` instead.

  Example: `nose2.plugins.result.ResultReporter`

- Writing a plugin that handles exceptions

  If you just want to handle some exceptions as skips or failures instead of errors, see `nose2.plugins.outcomes.Outcomes`, which offers a simple way to do that. Otherwise, implement `setTestOutcome()` to change test outcomes.

  Example: `nose2.plugins.outcomes.Outcomes`

- Writing a plugin that adds detail to error reports

  Implement `testOutcome()` and put your extra information into `event.metadata`, then implement `outcomeDetail()` to extract it and add it to the error report.

  Examples: `nose2.plugins.buffer.OutputBufferPlugin`, `nose2.plugins.logcapture.LogCapture`

- Writing a plugin that loads tests from files other than python modules

  Implement `handleFile()`.

  Example: `nose2.plugins.doctests.DocTestLoader`

- Writing a plugin that loads tests from python modules

  Implement at least `loadTestsFromModule()`.

  Warning: One thing to beware of here is that if you return tests a
  or instances of a testcase class that is defined *anywhere* but the m
  `nose2.util.transplant_class()` to make the test case class ap
  ule. Otherwise, module-level fixtures will not work for that test, and ma
  if there are no test cases that are or appear to be defined there.

- Writing a plugin that prints a report

  Implement `beforeErrorList()`, `beforeSummaryReport()` or `afterSummaryReport()`

  Example: `nose2.plugins.prof.Profiler`

- Writing a plugin that selects or rejects tests

  Implement `matchPath` or `getTestCaseNames`.

Example: nose2.plugins.loader.parameters.Parameters

2.2 Documenting plugins

You should do it. Nobody will use your plugins if you don't. Or if they do use them, they will curse you whenever things go wrong.

One easy way to document your plugins is to use nose2's Sphinx extension, which provides an `autoplugin` directive that will produce decent reference documentation from your plugin classes.

To use it, add 'nose2.sphinxext' to the `extensions` list in the `conf.py` file in your docs directory.

Then add an `autoplugin` directive to an rst file, like this:

```
.. autoplugin :: mypackage.plugins.PluginClass
```

This will produce output that includes the config vars your plugin loads in `__init__`, as well as any command line options your plugin registers. This is why you *really* should extract config vars and register command-line options in `__init__`.

The output will also include an `autoclass` section for your plugin class, so you can put more narrative documentation in the plugin's docstring for users to read.

Of course you can, and should, write some words before the reference docs explaining what your plugin does and how to use it. You can put those words in the rst file itself, or in the docstring of the module where your plugin lives.

2.3 Event reference

class nose2.events.**CommandLineArgsEvent**(*args*, ***kw*)
Event fired after parsing of command line arguments.

Plugins can respond to this event by configuring themselves or other plugins or modifying the parsed arguments.

Note: Many plugins register options with callbacks. By the time this event fires, those callbacks have already fired. So you can't use this event to reliably influence all plugins.

---

args
Args object returned by argparse.

class nose2.events.**CreateTestsEvent**(*loader*, *testNames*, *module*, ***kw*)
Event fired before test loading.

Plugins can take over test loading by returning a test suite and setting `handled` on this event.

loader
Test loader instance

names
List of test names. May be empty or None.

module
Module to load from. May be None. If not None, names should be considered relative to this module.

class nose2.events.**DescribeTestEvent**(*test*, *description=None*, *errorList=False*, ***kw*)
Event fired to get test description.

test
The test case

description
Description of the test case. Plugins can set this to change how tests are described in output to users.

errorList
Is the event fired as part of error list output?

class nose2.events.**Event**(***metadata*)
Base class for all events.

metadata
Storage for arbitrary information attached to an event.

handled
Set to True to indicate that a plugin has handled the event, and no other plugins or core systems should process it further.

version
Version of the event API. This will be incremented with each release of nose2 that changes the API.

version = '0.4'

class nose2.events.**GetTestCaseNamesEvent**(*loader*, *testCase*, *isTestMethod*, ***kw*)
Event fired to find test case names in a test case.

Plugins may return a list of names and set `handled` on this event to force test case name selection.

loader
Test loader instance

testCase
The `unittest.TestCase` instance being loaded.

testMethodPrefix
Set this to change the test method prefix. Unless set by a plugin, it is None.

extraNames
A list of extra test names to load from the test case. To cause extra tests to be loaded from the test case, append the names to this list. Note that the names here must be attributes of the test case.

excludedNames
A list of names to exclude from test loading. Add names to this list to prevent other plugins from loading the named tests.

isTestMethod
Callable that plugins can use to examine test case attributes to determine whether nose2 thinks they are test methods.

class nose2.events.**HandleFileEvent**(*loader*, *name*, *path*, *pattern*, *topLevelDirectory*, ***kw*)
Event fired when a non-test file is examined.

Note: This event is *not* fired for python modules that match the test file pattern.

---

loader
Test loader instance

name
File basename

path
Full path to file

pattern
Current test file match pattern

topLevelDirectory
Top-level directory of the test run

extraTests
A list of extra tests loaded from the file. To load tests from a file without interfering with other plugins' loading activities, append tests to extraTests.

Plugins may set `handled` on this event and return a test suite to prevent other plugins from loading tests from the file. If any plugin sets `handled` to True, `extraTests` will be ignored.

class `nose2.events.`**LoadFromModuleEvent**(*loader*, *module*, ***kw*)
Event fired when a test module is loaded.

loader
Test loader instance

module
The module whose tests are to be loaded

extraTests
A list of extra tests loaded from the module. To load tests from a module without interfering with other plugins' loading activities, append tests to extraTests.

Plugins may set `handled` on this event and return a test suite to prevent other plugins from loading tests from the module. If any plugin sets `handled` to True, `extraTests` will be ignored.

class `nose2.events.`**LoadFromNameEvent**(*loader*, *name*, *module*, ***kw*)
Event fired to load tests from test names.

loader
Test loader instance

name
Test name to load

module
Module to load from. May be None. If not None, names should be considered relative to this module.

extraTests
A list of extra tests loaded from the name. To load tests from a test name without interfering with other plugins' loading activities, append tests to extraTests.

Plugins may set `handled` on this event and return a test suite to prevent other plugins from loading tests from the test name. If any plugin sets `handled` to True, `extraTests` will be ignored.

class `nose2.events.`**LoadFromNamesEvent**(*loader*, *names*, *module*, ***kw*)
Event fired to load tests from test names.

loader
Test loader instance

names
List of test names. May be empty or None.

module
Module to load from. May be None. If not None, names should be considered relative to this module.

extraTests
A list of extra tests loaded from the tests named. To load tests from test names without interfering with other plugins' loading activities, append tests to extraTests.

Plugins may set `handled` on this event and return a test suite to prevent other plugins from loading tests from the test names. If any plugin sets `handled` to True, `extraTests` will be ignored.

class `nose2.events.`**LoadFromTestCaseEvent**(*loader*, *testCase*, ***kw*)
Event fired when tests are loaded from a test case.

loader
Test loader instance

testCase
The `unittest.TestCase` instance being loaded.

extraTests
A list of extra tests loaded from the module. To load tests from a test case without interfering with other plugins' loading activities, append tests to extraTests.

Plugins may set `handled` on this event and return a test suite to prevent other plugins from loading tests from the test case. If any plugin sets `handled` to True, `extraTests` will be ignored.

class `nose2.events.`**MatchPathEvent**(*name*, *path*, *pattern*, ***kw*)
Event fired during file matching.

Plugins may return False and set `handled` on this event to prevent a file from being matched as a test file, regardless of other system settings.

path
Full path to the file

name
File basename

pattern
Current test file match pattern

class `nose2.events.`**ModuleSuiteEvent**(*loader*, *module*, *suite*, ***kw*)

class `nose2.events.`**OutcomeDetailEvent**(*outcomeEvent*, ***kw*)
Event fired to acquire additional details about test outcome.

outcomeEvent
A `nose2.events.TestOutcomeEvent` instance holding the test outcome to be described.

extraDetail
Extra detail lines to be appended to test outcome output. Plugins can append lines (of strings) to this list to include their extra information in the error list report.

class `nose2.events.`**PluginsLoadedEvent**(*pluginsLoaded*, ***kw*)
Event fired after all plugin classes are loaded.

pluginsLoaded
List of all loaded plugin classes

class `nose2.events.`**ReportSummaryEvent**(*stopTestEvent*, *stream*, *reportCategories*, ***kw*)
Event fired before and after summary report.

stopTestEvent
A `nose2.events.StopTestEvent` instance.

stream
The output stream. Plugins can set this to change or capture output.

reportCategories
Dictionary of report category and test events captured in that category. Default categories include 'errors', 'failures', 'skipped', 'expectedFails', and 'unexpectedSuccesses'. Plugins may add their own categories.

class nose2.events.**ReportTestEvent**(*testEvent*, *stream*, ***kw*)
Event fired to report a test event.

Plugins can respond to this event by producing output for the user.

testEvent
A test event. In most cases, a nose2.events.TestOutcomeEvent instance. For startTest, a nose2.events.StartTestEvent instance.

stream
The output stream. Plugins can set this to change or capture output.

class nose2.events.**ResultCreatedEvent**(*result*, ***kw*)
Event fired when test result handler is created.

result
Test result handler instance. Plugins may replace the test result by setting this attribute to a new test result instance.

class nose2.events.**ResultStopEvent**(*result*, *shouldStop*, ***kw*)
Event fired when a test run is told to stop.

Plugins can use this event to prevent other plugins from stopping a test run.

result
Test result

shouldStop
Set to True to indicate that the test run should stop.

class nose2.events.**ResultSuccessEvent**(*result*, *success*, ***kw*)
Event fired at end of test run to determine success.

This event fires at the end of the test run and allows plugins to determine whether the test run was successful.

result
Test result

success
Set this to True to indicate that the test run was successful. If no plugin sets the success to True, the test run fails.

class nose2.events.**RunnerCreatedEvent**(*runner*, ***kw*)
Event fired when test runner is created.

runner
Test runner instance. Plugins may replace the test runner by setting this attribute to a new test runner instance.

class nose2.events.**StartTestEvent**(*test*, *result*, *startTime*, ***kw*)
Event fired before a test is executed.

test
The test case

result
Test result

startTime
Timestamp of test start

class nose2.events.**StartTestRunEvent**(*runner*, *suite*, *result*, *startTime*, *executeTests*, ***kw*)
Event fired when test run is about to start.

Test collection is complete before this event fires, but no tests have yet been executed.

runner
Test runner

suite
Top-level test suite to execute. Plugins can filter this suite, or set event.suite to change which tests execute (or how they execute).

result
Test result

startTime
Timestamp of test run start

executeTests
Callable that will be used to execute tests. Plugins may set this attribute to wrap or otherwise change test execution. The callable must match the signature:

```
def execute(suite, result):
    ...
```

To prevent normal test execution, plugins may set `handled` on this event to True. When `handled` is true, the test executor does not run at all.

class nose2.events.**StopTestEvent**(*test*, *result*, *stopTime*, ***kw*)
Event fired after a test is executed.

test
The test case

result
Test result

stopTime
Timestamp of test stop

class nose2.events.**StopTestRunEvent**(*runner*, *result*, *stopTime*, *timeTaken*, ***kw*)
Event fired when test run has stopped.

runner
Test runner

result
Test result

stopTime
Timestamp of test run stop

timeTaken
Number of seconds test run took to execute

class nose2.events.**TestOutcomeEvent**(*test*, *result*, *outcome*, *exc_info=None*, *reason=None*, *expected=False*, *shortLabel=None*, *longLabel=None*, ***kw*)
Event fired when a test completes.

test
The test case

result
Test result

outcome
Description of test outcome. Typically will be one of 'error', 'failed', 'skipped', or 'passed'.

exc_info
If the test resulted in an exception, the tuple of (exception class, exception value, traceback) as returned by sys.exc_info(). If the test did not result in an exception, None.

reason
For test outcomes that include a reason (Skips, for example), the reason.

expected
Boolean indicating whether the test outcome was expected. In general, all tests are expected to pass, and any other outcome will have expected as False. The exceptions to that rule are unexpected successes and expected failures.

shortLabel
A short label describing the test outcome. (For example, 'E' for errors).

longLabel
A long label describing the test outcome (for example, 'ERROR' for errors).

Plugins may influence how the rest of the system sees the test outcome by setting `outcome` or `exc_info` or `expected`. They may influence how the test outcome is reported to the user by setting `shortLabel` or `longLabel`.

class `nose2.events.`**`UserInteractionEvent`**`(`***kw*`)`
Event fired before and after user interaction.

Plugins that capture stdout or otherwise prevent user interaction should respond to this event.

To prevent the user interaction from occurring, return False and set `handled`. Otherwise, turn off whatever you are doing that prevents users from typing/clicking/touching/psionics/whatever.

2.4 Hook reference

Note: Hooks are listed here in order of execution.

2.4.1 Pre-registration Hooks

pluginsLoaded(*self*, *event*)

Parameters **event** – `nose2.events.PluginsLoadedEvent`

The `pluginsLoaded` hook is called after all config files have been read, and all plugin classes loaded. Plugins that register automatically (those that call `nose2.events.Plugin.register()` in __init__ or have `always-on = True` set in their config file sections) will have already been registered with the hooks they implement. Plugins waiting for command-line activation will not yet be registered.

Plugins can use this hook to examine or modify the set of loaded plugins, inject their own hook methods using `nose2.events.PluginInterface.addMethod()`, or take other actions to set up or configure themselves or the test run.

Since `pluginsLoaded` is a pre-registration hook, it is called for *all plugins* that implement the method, whether they have registered or not. Plugins that do not automatically register themselves should limit their actions in this hook to configuration, since they may not actually be active during the test run.

handleArgs(*self*, *event*)

Parameters **event** – nose2.events.CommandLineArgsEvent

The handleArgs hook is called after all arguments from the command line have been parsed. Plugins can use this hook to handle command-line arguments in non-standard ways. They should not use it to try to modify arguments seen by other plugins, since the order in which plugins execute in a hook is not guaranteed.

Since handleArgs is a pre-registration hook, it is called for *all plugins* that implement the method, whether they have registered or not. Plugins that do not automatically register themselves should limit their actions in this hook to configuration, since they may not actually be active during the test run.

2.4.2 Standard Hooks

These hooks are called for registered plugins only.

createTests (*self*, *event*)

Parameters **event** – A nose2.events.CreateTestsEvent instance

Plugins can take over test loading by returning a test suite and setting event.handled to True.

loadTestsFromNames (*self*, *event*)

Parameters **event** – A nose2.events.LoadFromNamesEvent instance

Plugins can return a test suite or list of test suites and set event.handled to True to prevent other plugins from loading tests from the given names, or append tests to event.extraTests. Plugins can also remove names from event.names to prevent other plugins from acting on those names.

loadTestsFromName (*self*, *event*)

Parameters **event** – A nose2.events.LoadFromNameEvent instance

Plugins can return a test suite and set event.handled to True to prevent other plugins from loading tests from the given name, or append tests to event.extraTests.

handleFile (*self*, *event*)

Parameters **event** – A nose2.events.HandleFileEvent instance

Plugins can use this hook to load tests from files that are not python modules. Plugins may either append tests to event.extraTest, or, if they want to prevent other plugins from processing the file, set event.handled to True and return a test case or test suite.

matchPath (*self*, *event*)

Parameters **event** – A nose2.events.MatchPathEvent instance

Plugins can use this hook to prevent python modules from being loaded by the test loader or force them to be loaded by the test loader. Set event.handled to True and return False to cause the loader to skip the module. Set event.handled to True and return True to cause the loader to load the module.

loadTestsFromModule (*self*, *event*)

Parameters **event** – A nose2.events.LoadFromModuleEvent instance

Plugins can use this hook to load tests from test modules. To prevent other plugins from loading from the module, set event.handled and return a test suite. Plugins can also append tests to event.extraTests – usually that's what you want to do, since that will allow other plugins to load their tests from the module as well.

See also *this warning* about test cases not defined in the module.

loadTestsFromTestCase (*self*, *event*)

Parameters **event** – A nose2.events.LoadFromTestCaseEvent instance

Plugins can use this hook to load tests from a unittest.TestCase. To prevent other plugins from loading tests from the test case, set event.handled to True and return a test suite. Plugins can also append tests to event.extraTests – usually that's what you want to do, since that will allow other plugins to load their tests from the test case as well.

getTestCaseNames (*self*, *event*)

Parameters **event** – A nose2.events.GetTestCaseNamesEvent instance

Plugins can use this hook to limit or extend the list of test case names that will be loaded from a unittest.TestCase by the standard nose2 test loader plugins (and other plugins that respect the results of the hook). To force a specific list of names, set event.handled to True and return a list: this exact list will be the only test case names loaded from the test case. Plugins can also extend the list of names by appending test names to event.extraNames, and exclude names by appending test names to event.excludedNames.

runnerCreated (*self*, *event*)

Parameters **event** – A nose2.events.RunnerCreatedEvent instance

Plugins can use this hook to wrap, capture or replace the test runner. To replace the test runner, set event.runner.

resultCreated (*self*, *event*)

Parameters **event** – A nose2.events.ResultCreatedEvent instance

Plugins can use this hook to wrap, capture or replace the test result. To replace the test result, set event.result.

startTestRun (*self*, *event*)

Parameters **event** – A nose2.events.StartTestRunEvent instance

Plugins can use this hook to take action before the start of the test run, and to replace or wrap the test executor. To replace the executor, set event.executeTests. This must be a callable that takes two arguments: the top-level test and the test result.

To prevent the test executor from running at all, set event.handled to True.

startTest (*self*, *event*)

Parameters **event** – A nose2.events.StartTestEvent instance

Plugins can use this hook to take action immediately before a test runs.

reportStartTest (*self*, *event*)

Parameters **event** – A nose2.events.ReportTestEvent instance

Plugins can use this hook to produce output for the user at the start of a test. If you want to print to the console, write to event.stream. Remember to respect self.session.verbosity when printing to the console. To prevent other plugins from reporting to the user, set event.handled to True.

describeTest (*self*, *event*)

Parameters **event** – A nose2.events.DescribeTestEvent instance

Plugins can use this hook to alter test descriptions. To return a nonstandard description for a test, set event.description. Be aware that other plugins may have set this also!

setTestOutcome (*self*, *event*)

Parameters **event** – A nose2.events.TestOutcomeEvent instance

Plugins can use this hook to alter test outcomes. Plugins can event.outcome to change the outcome of the event, tweak, change or remove event.exc_info, set or clear event.expected, and so on.

testOutcome(*self*, *event*)

Parameters **event** – A nose2.events.TestOutcomeEvent instance

Plugins can use this hook to take action based on the outcome of tests. Plugins *must not* alter test outcomes in this hook: that's what setTestOutcome() is for. Here, plugins may only react to the outcome event, not alter it.

reportSuccess(*self*, *event*)

Parameters **event** – A nose2.events.LoadFromNamesEvent instance

Plugins can use this hook to report test success to the user. If you want to print to the console, write to event.stream. Remember to respect self.session.verbosity when printing to the console. To prevent other plugins from reporting to the user, set event.handled to True.

reportError(*self*, *event*)

Parameters **event** – A nose2.events.ReportTestEvent instance

Plugins can use this hook to report a test error to the user. If you want to print to the console, write to event.stream. Remember to respect self.session.verbosity when printing to the console. To prevent other plugins from reporting to the user, set event.handled to True.

reportFailure(*self*, *event*)

Parameters **event** – A nose2.events.ReportTestEvent instance

Plugins can use this hook to report test failure to the user. If you want to print to the console, write to event.stream. Remember to respect self.session.verbosity when printing to the console. To prevent other plugins from reporting to the user, set event.handled to True.

reportSkip(*self*, *event*)

Parameters **event** – A nose2.events.ReportTestEvent instance

Plugins can use this hook to report a skipped test to the user. If you want to print to the console, write to event.stream. Remember to respect self.session.verbosity when printing to the console. To prevent other plugins from reporting to the user, set event.handled to True.

reportExpectedFailure(*self*, *event*)

Parameters **event** – A nose2.events.ReportTestEvent instance

Plugins can use this hook to report an expected failure to the user. If you want to print to the console, write to event.stream. Remember to respect self.session.verbosity when printing to the console. To prevent other plugins from reporting to the user, set event.handled to True.

reportUnexpectedSuccess(*self*, *event*)

Parameters **event** – A nose2.events.ReportTestEvent instance

Plugins can use this hook to report an unexpected success to the user. If you want to print to the console, write to event.stream. Remember to respect self.session.verbosity when printing to the console. To prevent other plugins from reporting to the user, set event.handled to True.

reportOtherOutcome(*self*, *event*)

Parameters **event** – A nose2.events.ReportTestEvent instance

Plugins can use this hook to report a custom test outcome to the user. If you want to print to the console, write to event.stream. Remember to respect self.session.verbosity when printing to the console. To prevent other plugins from reporting to the user, set event.handled to True.

stopTest(*self*, *event*)

Parameters **event** – A nose2.events.StopTestEvent instance

Plugins can use this hook to take action after a test has completed running and reported its outcome.

stopTestRun (*self*, *event*)

Parameters **event** – A nose2.events.StopTestRunEvent instance

Plugins can use this hook to take action at the end of a test run.

afterTestRun (*self*, *event*)

Parameters **event** – A nose2.events.StopTestRunEvent instance

---

Note: New in version 0.2

Plugins can use this hook to take action *after* the end of a test run, such as printing summary reports like the builtin result reporter plugin nose2.plugins.result.ResultReporter.

resultStop (*self*, *event*)

Parameters **event** – A nose2.events.ResultStopEvent instance

Plugins can use this hook to *prevent* other plugins from stopping a test run. This hook fires when something calls nose2.result.PluggableTestResult.stop(). If you want to prevent this from stopping the test run, set event.shouldStop to False.

beforeErrorList (*self*, *event*)

Parameters **event** – A nose2.events.ReportSummaryEvent instance

Plugins can use this hook to output or modify summary information before the list of errors and failures is output. To modify the categories of outcomes that will be reported, plugins can modify the event.reportCategories dictionary. Plugins can set, wrap or capture the output stream by reading or setting event.stream. If you want to print to the console, write to event.stream. Remember to respect self.session.verbosity when printing to the console.

outcomeDetail (*self*, *event*)

Parameters **event** – A nose2.events.OutcomeDetailEvent instance

Plugins can use this hook to add additional elements to error list output. Append extra detail lines to event.extraDetail; these will be joined together with newlines before being output as part of the detailed error/failure message, after the traceback.

beforeSummaryReport (*self*, *event*)

Parameters **event** – A nose2.events.ReportSummaryEvent instance

Plugins can use this hook to output or modify summary information before the summary lines are output. To modify the categories of outcomes that will be reported in the summary, plugins can modify the event.reportCategories dictionary. Plugins can set, wrap or capture the output stream by reading or setting event.stream. If you want to print to the console, write to event.stream. Remember to respect self.session.verbosity when printing to the console.

wasSuccessful (*self*, *event*)

Parameters **event** – A nose2.events.ResultSuccessEvent instance

Plugins can use this hook to mark a test run as successful or unsuccessful. If not plugin marks the run as successful, the default state is failure. To mark a run as successful, set event.success to True. Be ware that other plugins may set this attribute as well!

afterSummaryReport (*self*, *event*)

Parameters **event** – A nose2.events.ReportSummaryEvent instance

Plugins can use this hook to output a report to the user after the summary line is output. If you want to print to the console, write to event.stream. Remember to respect self.session.verbosity when printing to the console.

2.4.3 User Interaction Hooks

These hooks are called when plugins want to interact with the user.

beforeInteraction(*event*)

Parameters **event** – A nose2.events.UserInteractionEvent

Plugins should respond to this hook by getting out of the way of user interaction, if the need to, or setting event.handled and returning False, if they need to but can't.

afterInteraction(*event*)

Parameters **event** – A nose2.events.UserInteractionEvent

Plugins can respond to this hook by going back to whatever they were doing before the user stepped in and started poking around.

2.5 Session reference

2.5.1 Session

In nose2, all configuration for a test run is encapsulated in a Session instance. Plugins always have the session available as self.session.

class nose2.session.**Session**

Configuration session.

Encapsulates all configuration for a given test run.

argparse

An instance of argparse.ArgumentParser. Plugins can use this directly to add arguments and argument groups, but *must* do so in their __init__ methods.

pluginargs

The argparse argument group in which plugins (by default) place their command-line arguments. Plugins can use this directly to add arguments, but *must* do so in their __init__ methods.

hooks

The nose2.events.PluginInterface instance contains all available plugin methods and hooks.

plugins

The list of loaded – but not necessarily *active* – plugins.

verbosity

Current verbosity level. Default: 1.

startDir

Start directory of test run. Test discovery starts here. Default: current working directory.

topLevelDir

Top-level directory of test run. This directory is added to sys.path. Default: starting directory.

libDirs
Names of code directories, relative to starting directory. Default: ['lib', 'src']. These directories are added to sys.path and discovery if the exist.

testFilePattern
Pattern used to discover test module files. Default: test*.py

testMethodPrefix
Prefix used to discover test methods and functions: Default: 'test'.

unittest
The config section for nose2 itself.

configClass
alias of `Config`

get(*section*)
Get a config section.

Parameters **section** – The section name to retreive.

Returns instance of self.configClass.

loadConfigFiles(**filenames*)
Load config files.

Parameters **filenames** – Names of config files to load.

Loads all names files that exist into `self.config`.

loadPlugins(*modules=None*, *exclude=None*)
Load plugins.

Parameters **modules** – List of module names from which to load plugins.

loadPluginsFromModule(*module*)
Load plugins from a module.

Parameters **module** – A python module containing zero or more plugin classes.

prepareSysPath()
Add code directories to sys.path

registerPlugin(*plugin*)
Register a plugin.

Parameters **plugin** – A *nose2.events.Plugin* instance.

Register the plugin with all methods it implements.

2.5.2 Config

Configuration values loaded from config file sections are made available to plugins in `Config` instances. Plugins that set `configSection` will have a `Config` instance available as `self.config`.

class `nose2.config.`**Config**(*items*)
Configuration for a plugin or other entities.

Encapsulates configuration for a single plugin or other element. Corresponds to a `ConfigParser.Section` but provides an extended interface for extracting items as a certain type.

as_bool(*key*, *default=None*)
Get key value as boolean

1, t, true, on, yes and y (case insensitive) are accepted as True values. All other values are False.

as_float (*key*, *default=None*)
Get key value as float

as_int (*key*, *default=None*)
Get key value as integer

as_list (*key*, *default=None*)
Get key value as list.

The value is split into lines and returned as a list. Lines are stripped of whitespace, and lines beginning with # are skipped.

as_str (*key*, *default=None*)
Get key value as str

get (*key*, *default=None*)
Get key value

2.6 Plugin class reference

The plugin system in nose2 is based on the plugin system in unittest2's plugins branch.

2.6.1 Plugin base class

class nose2.events.**Plugin**
Base class for nose2 plugins

All nose2 plugins must subclass this class.

session
The nose2.session.Session under which the plugin has been loaded.

config
The nose2.config.Config representing the plugin's config section as loaded from the session's config files.

commandLineOption
A tuple of (short opt, long opt, help text) that defines a command line flag that activates this plugin. The short opt may be None. If defined, it must be a single upper-case character. Both short and long opt must *not* start with dashes.

Example:

```
commandLineOption = ('B', 'buffer-output', 'Buffer output during tests')
```

configSection
The name config file section to load into this plugin's config.

alwaysOn
If this plugin should automatically register itself, set alwaysOn to True. Default is False.

Note: Plugins that use config values from config files and want to use the nose2 sphinx extension to automatically generate documentation *must* extract all config values from `self.config` in `__init__`. Otherwise the extension will not be able to detect the config keys that the plugin uses.

---

addArgument(*callback*, *short_opt*, *long_opt*, *help_text=None*)
Add command-line option that takes one argument.

Parameters

- **callback** – Callback function to run when flag is seen. The callback will receive one argument.
- **short_opt** – Short option. Must be uppercase, no dashes.
- **long_opt** – Long option. Must not start with dashes
- **help_text** – Help text for users so they know what this flag does.

addFlag(*callback*, *short_opt*, *long_opt*, *help_text=None*)
Add command-line flag that takes no arguments

Parameters

- **callback** – Callback function to run when flag is seen. The callback will receive one empty argument.
- **short_opt** – Short option. Must be uppercase, no dashes.
- **long_opt** – Long option. Must not start with dashes
- **help_text** – Help text for users so they know what this flag does.

addMethods(**methods*)
Add new plugin methods to hooks registry

Any plugins that are already registered and implement a method added here will be registered for that method as well.

addOption(*callback*, *short_opt*, *long_opt*, *help_text=None*, *nargs=0*)
Add command-line option.

Parameters

- **callback** – Callback function to run when flag is seen. The callback will receive one argument. The "callback" may also be a list, in which case values submitted on the command line will be appended to the list.
- **short_opt** – Short option. Must be uppercase, no dashes.
- **long_opt** – Long option. Must not start with dashes
- **help_text** – Help text for users so they know what this flag does.
- **nargs** – Number of arguments to consume from command line.

register()
Register with appropriate hooks.

This activates the plugin and enables it to receive events.

2.6.2 Plugin interface classes

class nose2.events.**PluginInterface**
Definition of plugin interface.

Instances of this class contain the methods that may be called, and a dictionary of nose2.events.Hook instances bound to each method.

In a plugin, PluginInterface instance is typically available as self.session.hooks, and plugin hooks may be called on it directly:

```
event = events.LoadFromModuleEvent(module=the_module)
self.session.hooks.loadTestsFromModule(event)
```

preRegistrationMethods
Tuple of methods that are called before registration.

methods
Tuple of available plugin hook methods.

hookClass
Class to instantiate for each hook. Default: nose2.events.Hook.

addMethod(*method*)
Add a method to the available method.

This allows plugins to register for this method.

Parameters **method** – A method name

hookClass
alias of Hook

register(*method*, *plugin*)
Register a plugin for a method.

Parameters

- **method** – A method name
- **plugin** – A plugin instance

class nose2.events.**Hook**(*method*)
A plugin hook

Each plugin method in the nose2.events.PluginInterface is represented at runtime by a Hook instance that lists the plugins that should be called by that hook.

method
The name of the method that this Hook represents.

plugins
The list of plugin instances bound to this hook.

CHAPTER

THREE

DEVELOPER'S GUIDE

3.1 Contributing to nose2

3.1.1 Exhortation

Please do! nose2 cannot move forward without contributions from the testing community.

3.1.2 The Basics

nose2 is hosted on github. Our home there is https://github.com/nose-devs/nose2. We use github's issue tracking and collaboration tools *exclusively* for managing nose2's development. This means:

- Please report issues here: https://github.com/nose-devs/nose2/issues
- Please make feature requests in the same place.
- Please submit all patches as github pull requests.

3.1.3 Get started

The `bootstrap.sh` script in the root of the nose2 distribution can be used to get a new local clone up and running quickly. It requires that you have virtualenvwrapper installed. Run this script once to set up a nose2 virtualenv and install nose2's dependencies.

3.1.4 Coding Guidelines

Our style is pep8 except: for consistency with unittest, please use CamelCase for class names, methods, attributes and function parameters that map directly to class attributes.

Beyond style, the main rule is: *any patch that touches code must include tests.* And of course all tests must pass under all supported versions of Python.

Fortunately that's easy to check: nose2 uses tox to manage its test scenarios, so simply running `tox` in nose2's root directory will run all of the tests with all supported python versions. When your patch gets all green, send a pull request!

3.1.5 Merging Guidelines

The github Merge Button(tm) should be used only for trivial changes. Other merges, even those that can be automatically merged, should be merged manually, so that you have an opportunity to run tests on the merged changes before pushing them. When you merge manually, please use `--no-ff` so that we have a record of all merges.

Also, core devs should not merge their own work – again, unless it's trivial – without giving other developers a chance to review it. The basic workflow should be to do the work in a topic branch in your fork then post a pull request for that branch, whether you're a core developer or other contributor.

3.2 Internals

Reference material for things you probably only need to care about if you want to contribute to nose2.

3.2.1 nose2.main

class `nose2.main.`**`PluggableTestProgram`**(***kw*)

TestProgram that enables plugins.

Accepts the same parameters as `unittest.TestProgram`, but most of them are ignored as their functions are handled by plugins.

Parameters

- **module** – Module in which to run tests. Default: __main__
- **defaultTest** – Default test name. Default: None
- **argv** – Command line args. Default: sys.argv
- **testRunner** – *IGNORED*
- **testLoader** – *IGNORED*
- **exit** – Exit after running tests?
- **verbosity** – Base verbosity
- **failfast** – *IGNORED*
- **catchbreak** – *IGNORED*
- **buffer** – *IGNORED*
- **plugins** – List of additional plugin modules to load
- **excludePlugins** – List of plugin modules to exclude
- **extraHooks** – List of hook names and plugin *instances* to register with the session's hooks system. Each item in the list must be a 2-tuple of (hook name, plugin instance)

`sessionClass`

The class to instantiate to create a test run configuration session. Default: `nose2.session.Session`

`loaderClass`

The class to instantiate to create a test loader. Default: `nose2.loader.PluggableTestLoader`.

> **Warning:** Overriding this attribute is the only way to customize the test loader class. Passing a test loader to `__init__` does not work.

runnerClass
The class to instantiate to create a test runner. Default: nose2.runner.PluggableTestRunner.

> **Warning:** Overriding this attribute is the only way to customize the test runner class. Passing a test runner to __init__ does not work.

defaultPlugins
List of default plugin modules to load.

createTests()
Create top-level test suite

findConfigFiles(*cfg_args*)
Find available config files

handleArgs(*args*)
Handle further arguments.

Handle arguments parsed out of command line after plugins have been loaded (and injected their argument configuration).

handleCfgArgs(*cfg_args*)
Handle initial arguments.

Handle the initial, pre-plugin arguments parsed out of the command line.

loadPlugins()
Load available plugins

self.defaultPlugins and self.excludePlugins are passed to the session to alter the list of plugins that will be loaded.

This method also registers any (hook, plugin) pairs set in self.hooks. This is a good way to inject plugins that fall outside of the normal loading procedure, for example, plugins that need some runtime information that can't easily be passed to them through the configuration system.

loaderClass
alias of PluggableTestLoader

parseArgs(*argv*)
Parse command line args

Parses arguments and creates a configuration session, then calls createTests.

runTests()
Run tests

runnerClass
alias of PluggableTestRunner

sessionClass
alias of Session

setInitialArguments()
Set pre-plugin command-line arguments.

This set of arguments is parsed out of the command line before plugins are loaded.

nose2.main.**discover**(**args*, ***kwargs*)
Main entry point for test discovery.

Running discover calls nose2.main.PluggableTestProgram, passing through all arguments and keyword arguments **except module**: module is discarded, to force test discovery.

nose2.main.**main**
alias of PluggableTestProgram

3.2.2 nose2.compat

unittest/unittest2 compatibilty wrapper.

Anything internal to nose2 *must* import unittest from here, to be sure that it is using unittest2 when on older pythons.

Yes:

```
from nose2.compat import unittest
```

NO:

```
import unittest
```

NO:

```
import unittest2
```

3.2.3 nose2.exceptions

exception nose2.exceptions.**TestNotFoundError**
Exception raised when a named test cannot be found

3.2.4 nose2.loader

class nose2.loader.**PluggableTestLoader**(*session*)
Test loader that defers all loading to plugins

Parameters **session** – Test run session.

suiteClass
Suite class to use. Default: unittest.TestSuite.

discover(*start_dir=None*, *pattern=None*)
Compatibility shim for load_tests protocol.

failedImport(*name*)
Make test case representing a failed import.

failedLoadTests(*name*, *exception*)
Make test case representing a failed test load.

loadTestsFromModule(*module*)
Load tests from module.

Fires loadTestsFromModule() hook.

loadTestsFromName(*name*, *module=None*)
Load tests from test name.

Fires loadTestsFromName() hook.

loadTestsFromNames(*testNames*, *module=None*)
Load tests from test names.

Fires loadTestsFromNames() hook.

sortTestMethodsUsing(*name*)
Sort key for test case test methods.

suiteClass
alias of `TestSuite`

3.2.5 nose2.result

class `nose2.result.`**PluggableTestResult**(*session*)
Test result that defers to plugins.

All test outcome recording and reporting is deferred to plugins, which are expected to implement startTest, stopTest, testOutcome, and wasSuccessful.

Parameters **session** – Test run session.

shouldStop
When True, test run should stop before running another test.

addError(*test*, *err*)
Test case resulted in error.

Fires `setTestOutcome()` and `testOutcome()` hooks.

addExpectedFailure(*test*, *err*)
Test case resulted in expected failure.

Fires `setTestOutcome()` and `testOutcome()` hooks.

addFailure(*test*, *err*)
Test case resulted in failure.

Fires `setTestOutcome()` and `testOutcome()` hooks.

addSkip(*test*, *reason*)
Test case was skipped.

Fires `setTestOutcome()` and `testOutcome()` hooks.

addSuccess(*test*)
Test case resulted in success.

Fires `setTestOutcome()` and `testOutcome()` hooks.

addUnexpectedSuccess(*test*)
Test case resulted in unexpected success.

Fires `setTestOutcome()` and `testOutcome()` hooks.

startTest(*test*)
Start a test case.

Fires `startTest()` hook.

stop()
Stop test run.

Fires `resultStop()` hook, sets `self.shouldStop` to `event.shouldStop`.

stopTest(*test*)
Stop a test case.

Fires `stopTest()` hook.

wasSuccessful()
Was test run successful?

Fires `wasSuccessful()` hook, returns `event.success`.

3.2.6 nose2.runner

class nose2.runner.**PluggableTestRunner**(*session*)
Test runner that defers most work to plugins.

Parameters **session** – Test run session

resultClass
Class to instantiate to create test result. Default: `nose2.result.PluggableTestResult`.

resultClass
alias of `PluggableTestResult`

run(*test*)
Run tests.

Parameters **test** – A unittest TestSuite or TestClass.

Returns Test result

Fires `startTestRun()` and `stopTestRun()` hooks.

3.2.7 nose2.util

nose2.util.**ensure_importable**(*dirname*)
Ensure a directory is on sys.path

nose2.util.**exc_info_to_string**(*err*, *test*)
Format exception info for output

nose2.util.**format_traceback**(*test*, *err*)
Converts a sys.exc_info()-style tuple of values into a string.

nose2.util.**has_module_fixtures**(*test*)
Does this test live in a module with module fixtures?

nose2.util.**isgenerator**(*obj*)
is this object a generator?

nose2.util.**ispackage**(*path*)
Is this path a package directory?

nose2.util.**ln**(*label*, *char='-'*, *width=70*)
Draw a divider, with label in the middle.

```
>>> ln('hello there')
'---------------------------- hello there -----------------------------'
```

Width and divider char may be specified. Defaults are 70 and '-' respectively.

nose2.util.**module_from_name**(*name*)
Import module from name

nose2.util.**name_from_args**(*name*, *index*, *args*)
Create test name from test args

nose2.util.**name_from_path**(*path*)
Translate path into module name

nose2.util.**object_from_name**(*name*, *module=None*)
Import object from name

nose2.util.**parse_log_level**(*lvl*)
Return numeric log level given a string

nose2.util.**safe_decode**(*string*)
Safely decode a byte string into unicode

nose2.util.**test_from_name**(*name*, *module*)
Import test from name

nose2.util.**transplant_class**(*cls*, *module*)
Make class appear to reside in `module`.

Parameters

- **cls** – A class
- **module** – A module name

Returns A subclass of `cls` that appears to have been defined in `module`.

The returned class's `__name__` will be equal to `cls.__name__`, and its `__module__` equal to `module`.

nose2.util.**valid_module_name**(*path*)
Is path a valid module name?

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

n