

# Good Integration Practices

## Conventions for Python test discovery

pytest implements the following standard test discovery:

- If no arguments are specified then collection starts from testpaths (if configured) or the current directory. Alternatively, command line arguments can be used in any combination of directories, file names or node ids.
- Recurse into directories, unless they match norecursedirs.
- In those directories, search for `test_*.py` or `*_test.py` files, imported by their test package name.
- From those files, collect test items:
  - `test_` prefixed test functions or methods outside of class
  - `test_` prefixed test functions or methods inside Test prefixed test classes (without an `__init__` method)

For examples of how to customize your test discovery Changing standard (Python) test discovery.

Within Python modules, pytest also discovers tests using the standard unittest.TestCase subclassing technique.

## Choosing a test layout / import rules

pytest supports two common test layouts:

### Tests outside application code

Putting tests into an extra directory outside your actual application code might be useful if you have many functional tests or for other reasons want to keep tests separate from actual application code (often a good idea):

```

setup.py
mypkg/
  __init__.py
  app.py
  view.py
tests/
  test_app.py
  test_view.py
  ...

```

This way your tests can run easily against an installed version of mypkg.

Note that using this scheme your test files must have unique names, because pytest will import them as *top-level* modules since there are no packages to derive a full package name from. In other words, the test files in the example above will be imported as `test_app` and `test_view` top-level modules by adding `tests/` to `sys.path`.

If you need to have test modules with the same name, you might add `__init__.py` files to your tests folder and subfolders, changing them to packages:

```

setup.py
mypkg/
  ...
tests/
  __init__.py
  foo/
    __init__.py
    test_view.py
  bar/
    __init__.py
    test_view.py

```

Now pytest will load the modules as `tests.foo.test_view` and `tests.bar.test_view`, allowing you to have modules with the same name. But now this introduces a subtle problem: in order to load the test modules from the `tests` directory, pytest prepends the root of the repository to `sys.path`, which adds the side-effect that now `mypkg` is also importable. This is problematic if you are using a tool like `tox` to test your package in a virtual environment, because you want to test the *installed* version of your package, not the local code from the repository.

In this situation, it is strongly suggested to use a `src` layout where application root package resides in a sub-directory of your root:

```

setup.py
src/
  mypkg/
    __init__.py
    app.py
    view.py
tests/
  __init__.py
  foo/
    __init__.py
    test_view.py
  bar/
    __init__.py
    test_view.py

```

This layout prevents a lot of common pitfalls and has many benefits, which are better explained in this excellent [blog post by Ionel Cristian Mărieș](#).

## Tests as part of application code

Inlining test directories into your application package is useful if you have direct relation between tests and application modules and want to distribute them along with your application:

```

setup.py
mypkg/
  __init__.py
  app.py
  view.py
  test/
    __init__.py
    test_app.py
    test_view.py
    ...

```

In this scheme, it is easy to run your tests using the `--pyargs` option:

```
pytest --pyargs mypkg
```

pytest will discover where `mypkg` is installed and collect tests from there.

Note that this layout also works in conjunction with the `src` layout mentioned in the previous section.

### Note:

You can use Python3 namespace packages (PEP420) for your application but pytest will still perform test package name discovery based on the presence of `__init__.py` files. If you use one of the two recommended file system layouts above but leave away the `__init__.py` files from your directories it should just work on Python3.3 and above. From “inlined tests”, however, you will need to use absolute imports for getting at your application code.

### Note:

 v: latest ▼

If pytest finds a “a/b/test\_module.py” test file while recursing into the filesystem it determines the import name as follows:

- determine basedir: this is the first “upward” (towards the root) directory not containing an `__init__.py`. If e.g. both a and b contain an `__init__.py` file then the parent directory of a will become the basedir.
- perform `sys.path.insert(0, basedir)` to make the test module importable under the fully qualified import name.
- import `a.b.test_module` where the path is determined by converting path separators `/` into `.”` characters. This means you must follow the convention of having directory and file names map directly to the import names.

The reason for this somewhat evolved importing technique is that in larger projects multiple test modules might import from each other and thus deriving a canonical import name helps to avoid surprises such as a test module getting imported twice.

## Tox

For development, we recommend to use [virtualenv](#) environments and [pip](#) for installing your application and any dependencies as well as the pytest package itself. This ensures your code and dependencies are isolated from the system Python installation.

You can then install your package in “editable” mode:

```
pip install -e .
```

which lets you change your source code (both tests and application) and rerun tests at will. This is similar to running *python setup.py develop* or *conda develop* in that it installs your package using a symlink to your development code.

Once you are done with your work and want to make sure that your actual package passes all tests you may want to look into [tox](#), the virtualenv test automation tool and its [pytest support](#). Tox helps you to setup virtualenv environments with pre-defined dependencies and then executing a pre-configured test command with options. It will run tests against the installed package and not against your source code checkout, helping to detect packaging glitches.

## Integrating with setuptools / python setup.py test / pytest-runner

You can integrate test runs into your setuptools based project with the [pytest-runner](#) plugin.

Add this to setup.py file:

```
from setuptools import setup


setup(
    #...,
    setup_requires=['pytest-runner', ...],
    tests_require=['pytest', ...],
    #...,
)
```

And create an alias into setup.cfg file:

```
[aliases]
test=pytest
```

If you now type:

```
python setup.py test
```

this will execute your tests using `pytest - runner`. As this is a standalone version of `pytest` no prior installation whatsoever is required for calling the test command. You can also pass additional arguments to `pytest` such as your test directory or other options using `--addopts`.  [v: latest](#) ▼

You can also specify other pytest-ini options in your `setup.cfg` file by putting them into a `[tool:pytest]` section:

```
[tool:pytest]
addopts = --verbose
python_files = testing/**/*.py
```

## Manual Integration

If for some reason you don't want/can't use `pytest-runner`, you can write your own `setuptools` `Test` command for invoking `pytest`.

```
import sys

from setuptools.command.test import test as TestCommand

class PyTest(TestCommand):
    user_options = [('pytest-args=', 'a', "Arguments to pass to pytest")]

    def initialize_options(self):
        TestCommand.initialize_options(self)
        self.pytest_args = ''

    def run_tests(self):
        import shlex
        #import here, cause outside the eggs aren't loaded
        import pytest
        errno = pytest.main(shlex.split(self.pytest_args))
        sys.exit(errno)

setup(
    #...,
    tests_require=['pytest'],
    cmdclass = {'test': PyTest},
)
```

Now if you run:

```
python setup.py test
```

this will download `pytest` if needed and then run your tests as you would expect it to. You can pass a single string of arguments using the `--pytest-args` or `-a` command-line option. For example:

```
python setup.py test -a "--durations=5"
```

is equivalent to running `pytest --durations=5`.