# Testing Your Code

Testing your code is very important.

Getting used to writing testing code and running this code in parallel is now considered a good habit. Used wisely, this method helps you define more precisely your code's intent and have a more decoupled architecture.

Some general rules of testing:

- A testing unit should focus on one tiny bit of functionality and prove it correct.
- Each test unit must be fully independent. Each test must be able to run alone, and also within the test suite, regardless of the order that they are called. The implication of this rule is that each test must be loaded with a fresh dataset and may have to do some cleanup afterwards. This is usually handled by `setUp()` and `tearDown()` methods.
- Try hard to make tests that run fast. If one single test needs more than a few milliseconds to run, development will be slowed down or the tests will not be run as often as is desirable. In some cases, tests can't be fast because they need a complex data structure to work on, and this data structure must be loaded every time the test runs. Keep these heavier tests in a separate test suite that is run by some scheduled task, and run all other tests as often as needed.
- Learn your tools and learn how to run a single test or a test case. Then, when developing a function inside a module, run this function's tests frequently, ideally automatically when you save the code.
- Always run the full test suite before a coding session, and run it again after. This will give you more confidence that you did not break anything in the rest of the code.
- It is a good idea to implement a hook that runs all tests before pushing code to a shared repository.
- If you are in the middle of a development session and have to interrupt your work, it is a good idea to write a broken unit test about what you want to develop next. When coming back to work, you will have a pointer to where you were and get back on track faster.

- The first step when you are debugging your code is to write a new test pinpointing the bug. While it is not always possible to do, those bug catching tests are among the most valuable pieces of code in your project.
- Use long and descriptive names for testing functions. The style guide here is slightly different than that of running code, where short names are often preferred. The reason is testing functions are never called explicitly. `square()` or even `sqr()` is ok in running code, but in testing code you would have names such as `test_square_of_number_2()`, `test_square_negative_number()`. These function names are displayed when a test fails, and should be as descriptive as possible.
- When something goes wrong or has to be changed, and if your code has a good set of tests, you or other maintainers will rely largely on the testing suite to fix the problem or modify a given behavior. Therefore the testing code will be read as much as or even more than the running code. A unit test whose purpose is unclear is not very helpful in this case.
- Another use of the testing code is as an introduction to new developers. When someone will have to work on the code base, running and reading the related testing code is often the best thing that they can do to start. They will or should discover the hot spots, where most difficulties arise, and the corner cases. If they have to add some functionality, the first step should be to add a test to ensure that the new functionality is not already a working path that has not been plugged into the interface.

# The Basics

## unittest

**`unittest`** is the batteries-included test module in the Python standard library. Its API will be familiar to anyone who has used any of the JUnit/nUnit/CppUnit series of tools.

Creating test cases is accomplished by subclassing **`unittest.TestCase`**.

```
import unittest

def fun(x):
    return x + 1

class MyTest(unittest.TestCase):
    def test(self):
        self.assertEqual(fun(3), 4)
```

As of Python 2.7 unittest also includes its own test discovery mechanisms.

unittest in the standard library documentation

## Doctest

The **`doctest`** module searches for pieces of text that look like interactive Python sessions in docstrings, and then executes those sessions to verify that they work exactly as shown.

Doctests have a different use case than proper unit tests: they are usually less detailed and don't catch special cases or obscure regression bugs. They are useful as an expressive documentation of the main use cases of a module and its components. However, doctests should run automatically each time the full test suite runs.

A simple doctest in a function:

```
def square(x):
    """Return the square of x.

    >>> square(2)
    4
    >>> square(-2)
    4
    """

    return x * x

if __name__ == '__main__':
```

```
    import doctest
    doctest.testmod()
```

When running this module from the command line as in `python module.py`, the doctests will run and complain if anything is not behaving as described in the docstrings.

# Tools

## py.test

py.test is a no-boilerplate alternative to Python's standard unittest module.

```
$ pip install pytest
```

Despite being a fully-featured and extensible test tool, it boasts a simple syntax. Creating a test suite is as easy as writing a module with a couple of functions:

```
# content of test_sample.py
def func(x):
    return x + 1

def test_answer():
    assert func(3) == 5
```

and then running the *py.test* command:

```
$ py.test
========================== test session starts ============================
platform darwin -- Python 2.7.1 -- pytest-2.2.1
collecting ... collected 1 items

test_sample.py F

================================= FAILURES =================================
_____________________________ test_answer ______________________________

    def test_answer():
>       assert func(3) == 5
E       assert 4 == 5
E        +  where 4 = func(3)

test_sample.py:5: AssertionError
======================== 1 failed in 0.02 seconds =========================
```

is far less work than would be required for the equivalent functionality with the unittest module!

py.test

## Hypothesis

Hypothesis is a library which lets you write tests that are parameterized by a source of examples. It then generates simple and comprehensible examples that make your tests fail, letting you find more bugs with less work.

```
$ pip install hypothesis
```

For example, testing lists of floats will try many examples, but report the minimal example of each bug (distinguished exception type and location):

```
@given(lists(floats(allow_nan=False, allow_infinity=False), min_size=1))
def test_mean(xs):
    mean = sum(xs) / len(xs)
    assert min(xs) <= mean(xs) <= max(xs)
```

```
Falsifying example: test_mean(
    xs=[1.7976321109618856e+308, 6.102390043022755e+303]
)
```

Hypothesis is practical as well as very powerful and will often find bugs that escaped all other forms of testing. It integrates well with py.test, and has a strong focus on usability in both simple and advanced scenarios.

hypothesis

## tox

tox is a tool for automating test environment management and testing against multiple interpreter configurations.

```
$ pip install tox
```

tox allows you to configure complicated multi-parameter test matrices via a simple INI-style configuration file.

tox

## Unittest2

unittest2 is a backport of Python 2.7's unittest module which has an improved API and better assertions over the one available in previous versions of Python.

If you're using Python 2.6 or below, you can install it with pip:

```
$ pip install unittest2
```

You may want to import the module under the name unittest to make porting code to newer versions of the module easier in the future

```
import unittest2 as unittest

class MyTest(unittest.TestCase):
    ...
```

This way if you ever switch to a newer Python version and no longer need the unittest2 module, you can simply change the import in your test module without the need to change any other code.

unittest2

## mock

**`unittest.mock`** is a library for testing in Python. As of Python 3.3, it is available in the standard library.

For older versions of Python:

```
$ pip install mock
```

It allows you to replace parts of your system under test with mock objects and make assertions about how they have been used.

For example, you can monkey-patch a method:

```
from mock import MagicMock
thing = ProductionClass()
thing.method = MagicMock(return_value=3)
thing.method(3, 4, 5, key='value')

thing.method.assert_called_with(3, 4, 5, key='value')
```

To mock classes or objects in a module under test, use the `patch` decorator. In the example below, an external search system is replaced with a mock that always returns the same result (but only for the duration of the test).

```
    def mock_search(self):
        class MockSearchQuerySet(SearchQuerySet):
            def __iter__(self):
                return iter(["foo", "bar", "baz"])
        return MockSearchQuerySet()

    # SearchForm here refers to the imported class reference in myapp,
    # not where the SearchForm class itself is imported from
    @mock.patch('myapp.SearchForm.search', mock_search)
    def test_new_watchlist_activities(self):
        # get_search_results runs a search and iterates over the result
        self.assertEqual(len(myapp.get_search_results(q="fish")), 3)
```

Mock has many other ways you can configure it and control its behavior.

mock