Python Testing – Unittest

A unit testing framework

Overview

What is unittest?

- Unittest is a unit testing framework for Python inspired by unit testing frameworks for other programming languages such as JUnit.
- Included in the Python Standard Library

Key features

- Object-Oriented Design
- Test cases as sub-classes
- Custom pre/post condition methods for each test case (*Test Fixtures*)
- Multiple assertions (true, false, equal, raises error)
- Automatic discovery of test files in project (*Test Discovery*)
- Flagging tests to be skipped or expected to fail

Insights

- Easy to set up as it is included in the Python Standard Library
- Good baseline functionality, but no flashy or exceptional features
  - Other frameworks build off unittest

Installation

As unittest is included in Python's Standard Library, no additional installation is required aside from installing Python.

Installation Instructions

Windows

1. Download the latest stable release from the python website
2. Run the installer

   Select “Add python.exe to PATH”
3. Once installed, use the following command to verify that everything’s working using

   python --version or **python3 --version**

MacOS

1. Download the latest stable release from the python website
2. Run the installer
3. Once installed, use the following command to verify that everything’s working using:

   python --version or **python3 --version**

Installation Instructions

*Some distributions of Linux come with python pre-installed. If python is already installed (verify using the command **python --version** or **python3 --version**), then these instructions can be skipped.

Ubuntu Linux

1. Run **sudo apt-get update**
2. Run **sudo apt-get install python3**
3. Once installed, use the following command to verify that everything's working using

   python --version or **python3 --version**

Debian Linux

1. Run **sudo apt install python3**
2. Once installed, use the following command to verify that everything's working using

   python --version or **python3 --version**

Usage

Assertions

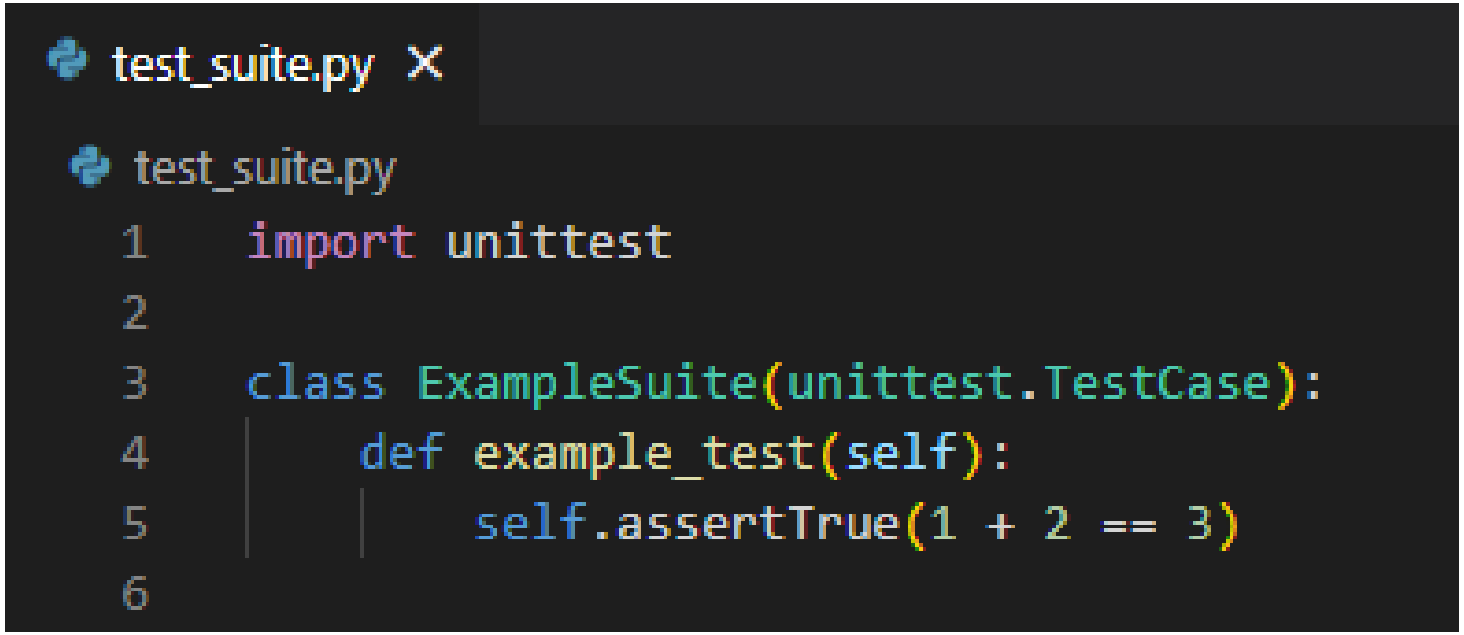
unittest relies on assertions to assess functionality.

The following table outlines common assertions in the framework. A list of all assertions can be found here.

Method Name	Checks
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a,b)
assertNotIsInstance(a, b)	not isinstance(a,b)
assertRaises(exc, fun, *args, **kwds)	fun(*args, **kwds) raises *exc*
assertWarns(warn, fun, *args, **kwds)	fun(*args, **kwds) raises *warn*
assertLogs(logger, level)	The **with** block logs on logger with minimum level

Basic Test Creation

1. Create a python file for your test suite
   - “test_suite.py” in example
2. Import unittest
   - Line 1 in example
3. Create a subclass for your test suite
   - Line 3 in example
4. Create methods within this class for each of your test cases
   - Line 4 in example
5. Within the methods, use one of the asserts to verify some functionality.
   - Line 5 in example
   - Calls to other methods can be included inside of the parenthesis



Running Tests

- Tests can be run from the unittest command line runner by calling the following function from inside the directory containing your test file:

  python –m unittest filename.py

- Replace filename.py with the name you used to create your test suite. If using the previous example, this command would be:

  python –m unittest test_suite.py

Using Test Discovery

- Tests can be automatically discovered and run by including a “test/” directory in your project and placing your testing files inside of this directory.
- If the above directory structure is followed, you can run unittest in discovery mode using the following commands:

  python –m unittest discover

  python –m unittest

  - The two commands above are equivalent. Calling unittest without arguments causes the test runner to use discovery mode.

Using Pre/Post Conditions

Pre-Conditions

1. Include a setUp method in your test suite

   This can be useful for setting up instance variables or objects before running tests

Post-Conditions

1. Include a tearDown method in your test suite

   This can be useful to remove instance variables or objects that were created in the setUp method.

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
```

Example from Python documentation for unittest

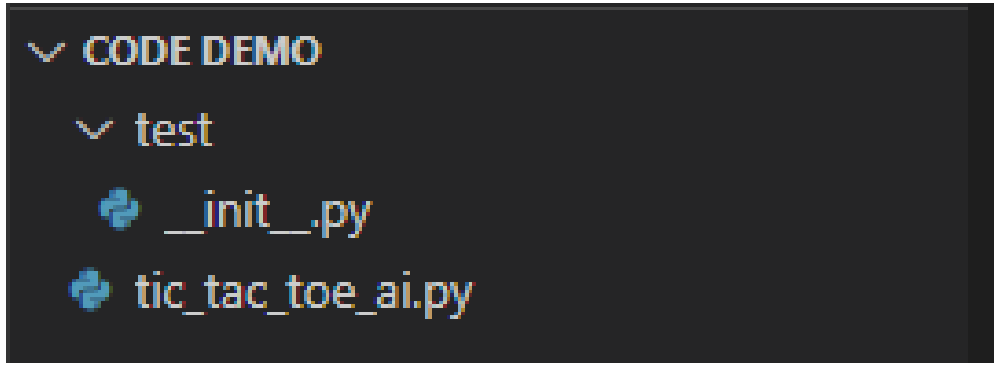
Code Demo

Python File Acquisition

1. Find a python file to test
    - A sample python file “tic_tac_toe_ai.py” is included in this module, and can be accessed by downloading it from the “View Python File” button

Directory Setup

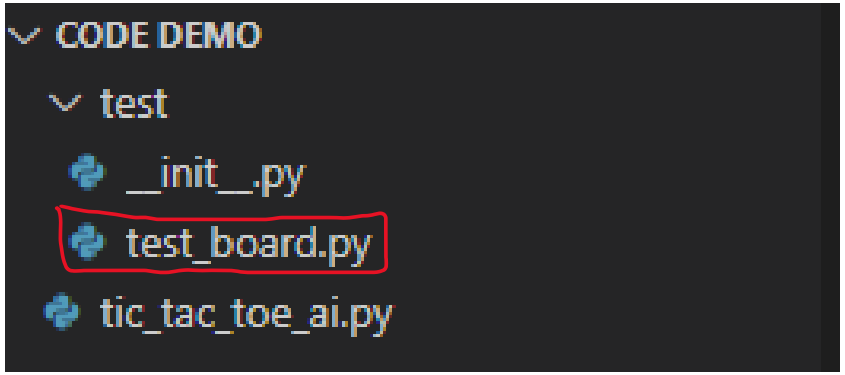
2. Place your python file(s) inside of a directory
3. Create a new directory alongside your python file(s) named “test”
4. Create a new file inside of the test directory called “__init__.py”



Test File Creation

5. Create a new python file inside of the test/ directory.

    5. a) The name of this python file should begin with “test_”



Test Case Creation

6. Inside of the file created in 5., create one or more test cases.

- Remember to import unittest and any methods from the python file you wish to test
- setUp methods can be helpful here to initialize any objects needed in the tests.

test_board.py

test > test_board.py

```python
# test/test_board.py
import unittest
from tic_tac_toe_ai import space_check, full_board_check

class TestBoard(unittest.TestCase):
    # Create an empty board for testing
    def setUp(self):
        self.board = [" "] * 10 # indexing for this program starts at 1
        self.avail = [" "] * 10

    # This test case checks if empty spaces are handled accurately
    def test_empty_space(self):
        result = space_check(self.board, 1)
        self.assertTrue(result)

    # This test case checks if occupied spaces are handled accurately
    def test_occupied_space(self):
        self.board[1] = "X"
        result = space_check(self.board, 1)
        self.assertFalse(result)

```

Running Tests

```
PS B:\Documents\OneDrive - University of Guelph\A F23\CIS 4150\Presentation\Code Demo> python3 -m unittest
.....
----------------------------------------------------------------------
Ran 5 tests in 0.001s

OK
PS B:\Documents\OneDrive - University of Guelph\A F23\CIS 4150\Presentation\Code Demo>
```

7. Running the test cases is as easy as running **python –m unittest** from the root directory of the project. This will run unittest in discovery mode.

```
PS B:\Documents\OneDrive - University of Guelph\A F23\CIS 4150\Presentation\Code Demo> python3 -m unittest test/test_board.py
.....
----------------------------------------------------------------------
Ran 5 tests in 0.001s

OK
PS B:\Documents\OneDrive - University of Guelph\A F23\CIS 4150\Presentation\Code Demo>
```

7. a) If you have one specific test file you'd like to run, you can pass it in as a param

i.e., **python –m unittest test/test_board.py**