CIS 1100

Unit Testing

Python
Fall 2024
University of Pennsylvania

Learning Objectives

- Be able to write unit tests in Python
- Be able to run unit tests in Python
- Be able to evaluate unit test output in Python

CIS 1100

Why Test Your Code?

Python
Fall 2024
University of Pennsylvania

Writing Programs That Are "Correct"

How do we decide if the code we've written is "correct"?

- We could just *think really hard* about what we've written
- We could use *formal verification techniques* to prove our code is correct
- We can systematically run our code and see what it does.
  - adding print statements
  - submitting code to an autograder and looking at outputs

Unit Testing

Unit Testing is the process of determining correctness for individual **units** of code by writing **test cases.**

- Units of code: functions, basically
- Test cases:
  - also functions
  - machinery for automatically determining if a piece of a program exhibited the correct behavior

Why Unit Testing?

- Writing functions allows us to develop large programs in small pieces
  - functions should have clearly defined *purpose* and *intended behavior*
- Easier to formalize correctness for small pieces rather than large programs
  - hard to answer *"is my Caesar Cipher correct?"*
  - easier to answer *"does my string-to-symbol conversion work properly in this case?"*

Limits of Unit Testing

- Our case for "correctness" is only as good as the tests that we write
- More units of code ➡️ more test cases to write
- Test cases are functions (code) themselves, which can themselves have bugs!

CIS 1100

What is a Unit Test?

Python
Fall 2024
University of Pennsylvania

Recipe for a Unit Test

How do we test our code to determine if it’s right?

- Identify the **INPUT**, possibly including any state variables
- Generate, manually or through means OUTSIDE of your code an **EXPECTED OUTPUT**
- Execute your code to get an **ACTUAL OUTPUT**
- Compare the expected and actual output

A Function to Test

Nested conditionals, no comments... 🤯

```
def find_max(a, b, c):
    if a > b:
        if a > c:
            return a
        else:
            return c
    else:
        if b > c:
            return b
        else:
            return a
```

Generating Tests

`find_max(a, b, c)` is a function that should return the largest of its three inputs.

- What is the *expected output* for `find_max` when called on inputs `3, 2, 1`?
- What is the *expected output* for `find_max` when called on inputs `1, 2, 3`?

In both cases: 3

Generating Tests

Test Case #1:

Input = `3, 2, 1`;

Expected output = `3`;

Actual output = `find_max(3, 2, 1)`

Test Case #2:

Input = `1, 2, 3`;

Expected output = `3`;

Actual output = `find_max(1, 2, 3)`

Evaluating Tests

The actual output is always determined by calling the function on the inputs (e.g. `actual = find_max(a, b, c)`)

- Test cases **pass** when the expected and actual outputs match
- Test cases **fail** when the expected and actual outputs differ

Is this a *passing* or *failing* test case?

Test Case #1: Input = `3, 2, 1`; Expected output = `3`; Actual output = `3`

Is this a *passing* or *failing* test case?

Test Case #2: Input = `1, 2, 3`; Expected output = `3`; Actual output = `1`

Evaluating Tests

Is this a *passing* or *failing* test case? **PASSING!** ✅

Test Case #1: Input = `3, 2, 1`; Expected output = `3`; Actual output = `3`

Is this a *passing* or *failing* test case? **FAILING!** ❗

Test Case #2: Input = `1, 2, 3`; Expected output = `3`; Actual output = `1`

Testing Is Like Potato Chips...

> "...they both contribute to my overall poor health.
> Also, you can't have just one." - Will McBurney

One test passing may have no bearing on another test passing! One test is not enough to decide if your implementation is bug-free.

- More tests is better:
  - more passing tests ➡️ more positive examples of your success
  - any failing tests ➡️ signposts to *faults* in your code

Why Does This Test Pass While That One Fails?

Test 1 does not cover/execute the underlying **fault** in the code.

- A **fault** is a particular defect in the code, or bug.
- Test 1 was still important for building confidence that the program works
- Test 2 is very important for identifying a bug in the program to fix

Test Case #1: Input = `{3,2,1}`; Expected output = `3`; Actual output = `3`
Test Case #2: Input = `{1,2,3}`; Expected output = `3`; Actual output = `1`

CIS 1100

Writing Unit Tests in Python

Python
Fall 2024
University of Pennsylvania

`unittest`: Our Testing Module

Python comes with `unittest` built in

- allows us to write unit tests that produce descriptive outputs when passing & failing
- comes with machinery for finding & running all unit tests in your project

Basic `unittest` Scaffold

```
import unittest
import other_module

class TestTextAnalysis(unittest.TestCase):

    def test_upper(self):
        to_modify = "cis"                   # INPUT
        expected = "CIS"                    # EXPECTED
        actual = to_modify.upper()          # ACTUAL
        self.assertEqual(expected, actual)  # ASSERTION

    def test_is_palindrome(self):
        a_palindrome = "racecar"
        expected = True
        actual = other_module.is_palindrome(a_palindrome)
        self.assertEqual(expected, actual)

if __name__ == '__main__':
    unittest.main()
```

Ingredients of `unittest` Test Suites

- A *test suite* is a collection of unit tests that should all be run together
  - i.e. all of the tests that you write for a project
- A *test case* is a function representing a unit test
  - Consists of an input, expected output, and actual output
  - Includes an *assertion statement* that asks Python to verify that something is true

Test Suites & Classes

- The `class` is a fixture of *object oriented programming*
  - more on this later
- for now, just know to group individual test cases together in a `class`
  - i.e. a `class` can be a test suite
  - name of `class` must be followed by `(unittest.TestCase)`
- all test cases (functions) must have `self` as their only input

```
class TestMyCode(unittest.TestCase):    # class as test suite

    def test_one(self):                 # one individual test case indented within class
        ...

    def test_two(self):                 # another test case
        ...
```

Zooming out from the Class

```
import unittest
import other_module

class TestMyCode(unittest.TestCase):

    def test_one(self):
        ...

    def test_two(self):
        ...

if __name__ == '__main__':
    unittest.main()
```

- Must import `unittest` to do unit testing
- `other_module` is a generic name for some other file containing code we want to test
- `if __name__ ...` allows the file containing this test suite to be executed directly

Assertions

Assertions are functions from the `unittest` module that:

- verify whether come condition of correctness is true
- dictate whether a test passes or fails
- help produce useful messages as output when running all tests

Assertions

Kind of Assertion	Meaning
`self.assertEqual(expected, actual)`	Test passes when `expected == actual`
`self.assertTrue(result)`	Test passes when `result` is `True`
`self.assertIsNone(result)`	Shorthand for `self.assertEqual(None, result)`

- Many more, too, including `assertNotEqual`, `assertFalse`, `assertIsNotNone`...
- Can also include a `str` as a final argument to provide a message that should be printed if the test fails
  - e.g. `self.assertTrue(result, "all odd numbers should produce False value")`

Back to `find_max`

`my_code.py`:

```
def find_max(a, b, c):
    if a > b:
        if a > c:
            return a
        else:
            return c
    else:
        if b > c:
            return b
        else:
            return a
```

Back to `find_max`

`test_my_code.py`:

```python
import unittest
import my_code

class TestFindMax(unittest.TestCase):

    def test_find_max_3_2_1(self):
        a, b, c = 3, 2, 1                       # Setting inputs
        expected = 3                            # Expected output
        actual = my_code.find_max(a, b, c)      # Actual output
        self.assertEqual(expected, actual)      # Assertion

    def test_find_max_1_2_3(self):
        a, b, c = 1, 2, 3                       # Setting inputs
        expected = 3                            # Expected output
        actual = my_code.find_max(a, b, c)      # Actual output
        self.assertEqual(expected, actual)      # Assertion

if __name__ == '__main__':
    unittest.main()
```

Running Tests

Write `python test_my_code.py` in the terminal:

```
codio@equatormaxwell-octobertina:~/workspace$ python test_my_code.py
F.
======================================================================
FAIL: test_find_max_1_2_3 (__main__.TestFindMax.test_find_max_1_2_3)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/codio/workspace/test_my_code.py", line 16, in test_find_max_1_2_3
    self.assertEqual(expected, actual) # Assertion
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 3 != 1

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Understanding the Output

Summary:

```
F.
```

- One character per test
  - `.` means a passing test
  - `F` means a failing test
  - `E` (not pictured) means a test that crashed

So:

- Two tests were run
  - one test passed
  - one test failed

Understanding the Output

Each error or failure will come with a more verbose description of what went wrong:

```
======================================================================
FAIL: test_find_max_1_2_3 (__main__.TestFindMax.test_find_max_1_2_3)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/codio/workspace/test_my_code.py", line 16, in test_find_max_1_2_3
    self.assertEqual(expected, actual) # Assertion
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 3 != 1
```

"`test_find_max_1_2_3` failed because we expected `3` but actually got `1`."

Fixing `find_max`

We know there's an issue where `find_max(1, 2, 3)` returns `1` instead of `3`...

```
def find_max(a, b, c):
    if a > b:
        if a > c:
            return a
        else:
            return c
    else:
        if b > c:
            return b
        else:
            return a
```

Fixing `find_max`

We know there's an issue where `find_max(1, 2, 3)` returns `1` instead of `3`...

```
def find_max(1, 2, 3):
    if 1 > 2:          # skip
        if 1 > 3:
            return 1
        else:
            return 3
    else:
        if 2 > 3:      # skip
            return 2
        else:          # execute...
            return 1     # uh oh!
```

Fixing `find_max`

```
def find_max(a, b, c):
    if a > b:
        if a > c:
            return a
        else:
            return c
    else:
        if b > c:
            return b
        else:
            return c
```

Running the Tests Again

```
codio@equatormaxwell-octobertina:~/workspace$ python test_my_code.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

- Two dots ➡️ two passing test cases!
- `OK` in the output ➡️ no failures or errors to report

Shortcut for Running All Tests

To run all test cases in all test suites in all files with names starting with `test`, just use:

```
python -m unittest
```

Running All Tests

I have `test_my_code.py` (the two tests we've seen) and `test_other_stuff.py` (three tests you haven't seen):

```
codio@equatormaxwell-octobertina:~/workspace$ python -m unittest
..EF.
======================================================================
ERROR: test_this_crashes (test_other_stuff.TestOtherStuff.test_this_crashes)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/codio/workspace/test_other_stuff.py", line 9, in test_this_crashes
    self.assertTrue(33 > "yes") # Assertion
                    ^^^^^^^^^^
TypeError: '>' not supported between instances of 'int' and 'str'

======================================================================
FAIL: test_this_fails (test_other_stuff.TestOtherStuff.test_this_fails)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/codio/workspace/test_other_stuff.py", line 12, in test_this_fails
    self.fail()
AssertionError: None

----------------------------------------------------------------------
Ran 5 tests in 0.002s

FAILED (failures=1, errors=1)
```