CIS 1100

Unit Testing

Python
Fall 2024
University of Pennsylvania

Recipe for a Unit Test

To test a function (e.g. `encrypt`), we always need:

- **The Input(s)**: `"ET TU, BRUTE?"` and `"G"`
- **The Expected Output**: `KZ ZA, HXAZK?`
- **The Actual Output**: gotten from running the `encrypt` function
- **Compariing the Expected and Actual Output**: Having you look at the output and making sure it looks correct.

Edge Cases

It is important to have a wide variety of tests to cover all cases.

- If we pass a test, we only know it works for those cases (those particular inputs)
- Cases that are not tested may not work
- We often want to test a variety of inputs and any "edge cases" (cases that are not common or obvious, but still should be handled correctly)

Practice (C12)

Consider the following function header:

```
# Given a string, returns a dictionary that contains all the words (split on whitespace)
# mapped onto how many times that word shows up in the string.
def get_word_counts(string):
```

Come up 3 different test cases, choose one of them and write the python test code for it

Practice (C14)

Now that we have written some tests, lets write the function itself.

Sometimes it actually helps to write the tests first, it can help with thinking about all the cases the code will have to handle.

```
# Given a string, returns a dictionary that contains all the words (split on whitespace)
# mapped onto how many times that word shows up in the string.
def get_word_counts(string):
```

Hint: `dict[key] = value` and `split()` may be useful

Why use `unittest` instead of just printing?

If we were able to get similar affects by just printing? Why use unittest?

- Helps keep the code organized: Which stuff is for testing and which stuff is part of the actual program
- Unlike what we did in Caesar, the test can check the output for us

Activity: Passing & Failing

```
def my_add(a, b):
    """Return the sum of a and b."""
    return a + b
```

How many of these tests pass? (Assume that we imported `my_add` properly...) **(S7)**

```
def test_three_plus_four(self):
    a = 3
    b = 4
    self.assertEqual(7, my_add(a, b))

def test_neg_three_plus_four(self):
    a = -3
    b = 4
    self.assertEqual(-1, my_add(a, b))
```

```
def test_pt_two_plus_pt_one(self):
    a = 0.2
    b = 0.1
    self.assertEqual(0.3, my_add(a, b))

def test_neg_zero_plus_zero(self):
    a = 0.0
    b = 0
    self.assertEqual(0, my_add(a, b))
```

Two Takeaways

1. Failing test cases can happen because of an error in the *test case* rather than a fault in the underlying code
2. `float` values can have small amounts of rounding error. When writing unit tests for non-integer numeric values, use `assertAlmostEqual(expected, actual, places=7)`
    i. The setting for the `places` kwarg chooses which decimal place to round the difference between `expected` and `actual` to.
    ii. `places=10` is more "strict" than `places=4` is more "strict" than `places=0`

```
def test_pt_two_plus_pt_one_correct(self):
    a = 0.2
    b = 0.1
    self.assertAlmostEqual(0.3, my_add(a, b))
```

Towards HW04...

Homework 4 is **Food Recommender**:

- Write a function that reads a *.CSV* file and transforms it into a `dict` mapping restaurant names to tuples of restaurant data
- Write a bunch of functions that find restaurants that match certain conditions in the dictionary
- Write a bunch of functions that aggregate information about all of the restaurants or all of the restaurants that meet a certain condition
- **TEST** these functions

Picking Up on `books.txt`

From last Wednesday:

- read a structured `.txt` file that contained information about a bunch of books into a `dict` mapping book names to tuples of book data

Next:

- Find all books by a given author
- Find the highest rated book released in a year
- **TEST** these

Last Time: Process Books

Returns a dictionary mapping book titles to tuples of book information.

```
def process_book_file(filename):
    book_file = open(filename, 'r')
    num_books = int(book_file.readline().strip())
    d = dict()
    for _ in range(num_books):
        title = book_file.readline().strip()
        author = book_file.readline().strip()
        year, pages, rating = book_file.readline().strip().split()
        d[title] = (author, year, pages, rating)

    book_file.close()
    return d
```

Activity: How to Test

Outside of just printing the output and looking at it, how could we test it?

Activity: How to Test

Outside of just printing the output and looking at it, how could we test it?

- Could check that the output dictionary is *exactly* what we expect
  - Hard for big files like `books.txt`
  - Easier if we create a smaller input file
- Check that the output dictionary has the right size (easier)
- Check that the output dictionary contains *some arbitrary* elements of `books.txt` that you know should be there

Activity: How to Test

Outside of just printing the output and looking at it, how could we test it?

- Could check that the output dictionary is *exactly* what we expect
  - Hard for big files like `books.txt`
  - Easier if we create a smaller input file
- Check that the output dictionary has the right size (easier)
- Check that the output dictionary contains *some arbitrary* elements of `books.txt` that you know should be there

In **(L11)**, describe a test case (with expected and actual output) for:

- a "typical" input
- an "edge case" input

Next up: Books By Author

Return a set of the titles of all of the books that were written by a given author.

```
def books_by_author(books, author):
```

First, let's think about testing. Generating expected results can be challenging, but it's how you decide if you're right or not.

In **(S8)**, write the value for `expected` that would make this test pass. Do some sleuthing! (In Codio, Find -> Find might help)

```
def test_books_by_author_ginzburg(self):
    d = book_recommender.process_book_file("books.txt")
    author = "Natalia Ginzburg"
    result_set = book_recommender.books_by_author(d, author)
    actual = len(result_set)
    expected = ???
    self.assertEqual(expected, actual)
```

Next up: Books By Author

In **(S9)**, write the value for `expected` that would make this test pass. Do some sleuthing! (In Codio, Find -> Find might help)

```
def test_books_by_author_zambreno(self):
    d = book_recommender.process_book_file("books.txt")
    author = "Kate Zambreno"
    result_set = book_recommender.books_by_author(d, author)
    self.assertEqual(1, len(result_set))

    expected = ???
    self.assertTrue(expected in result_set)
```

Next up: Implement Books By Author

Return a set of the titles of all of the books that were written by a given author.

```
def books_by_author(books, author):
```

Not vital, but can you do it in one line with a comprehension?