Section 8

Testing

Logistics

HW5 Part 1 due Monday 11/21

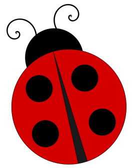
Testing

A way to make sure your code is behaving how you expect

The Rules of Testing

- Do it early and often
    - Catch bugs before they have a chance to hide!
- Be systematic
    - If you thrash about randomly, the bugs will hide in the corner and come out when you’re gone!!

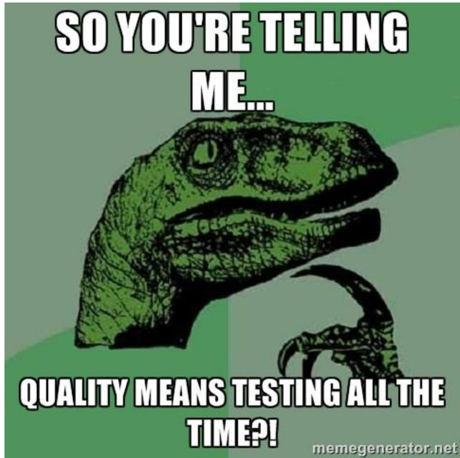
stinky ->



Unit Testing

Test focuses on one function (or part of a function) at a time

1. Write tests *before* you start designing your function. (Try to, anyways -- there are times where it's not plausible)
    a. Choose input data
    b. Define expected output
2. Run tests often



Choosing your tests

We want good coverage by tests, but we also don’t want to overdo it.

For example, we want to write a function that takes an integer and calculates a factorial. Do we need to do this for every possible integer? Probably not.

- Identify sets of inputs with the same behavior
- Try at least one input from each set

Choosing your tests

We want to test a factorial function. What are some sets of inputs with similar behavior?

- Small positive numbers
- Large positive numbers
- 0

What would be some examples of regular cases and edge cases?

More Testing Rules

Run tests as often as possible

Run tests every time you change something big

Keep all old tests and keep running them! Don’t delete tests after you’ve passed them, you could accidentally do something that could break your old test!

What about unexpected behavior?

Say we don’t want our function to calculate the factorial of negative numbers

You could specify what happens in the case for negative numbers in docstring. You could specify what happens in this case

- ex: returns None if given a negative integer

You could also say that the input must be a non-negative integer. If weird stuff happens with negative numbers, that’s not on you. You said that the input needed to be non-negative!

Read the Error Message

Call stack or traceback

```
Traceback (most recent call last):
    File "nx_error.py", line 41, in <module>
        print(friends_of_friends(rj, myval))
    File "nx_error.py", line 30, in friends_of_friends
        f = friends(graph, user)
    File "nx_error.py", line 25, in friends
        return set(graph.neighbors(user))#
    File "/Library/Frameworks/…/graph.py", line 978, in neighbors
        return list(self.adj[n])
TypeError: unhashable type: 'list'
```

First function that was called (`<module>` means the interpreter)

Second function that was called

…

Last function that was called (this one suffered an error)

The error message. You need to understand:
- the literal meaning of the error
- the underlying problems certain errors tend to suggest

Assertions

“How to deal with the fact the world doesn’t always work right”

```
assert <condition>,<error message>
```

example:

```
observed = factorial(0)
expected = 1
assert observed == expected, “Expected ” + str(expected)+ “, but saw ” + str(observed)
```

Example of a failure:

```
>> AssertionError: Expected 1, but saw 0
```

When to use Assertions

- When you’re writing code
- When you’re testing
- Checking if input to a function is correct
    - For example, you could assert that the input to the factorial function is a number, or is not less than 0

Great for helping you find errors early

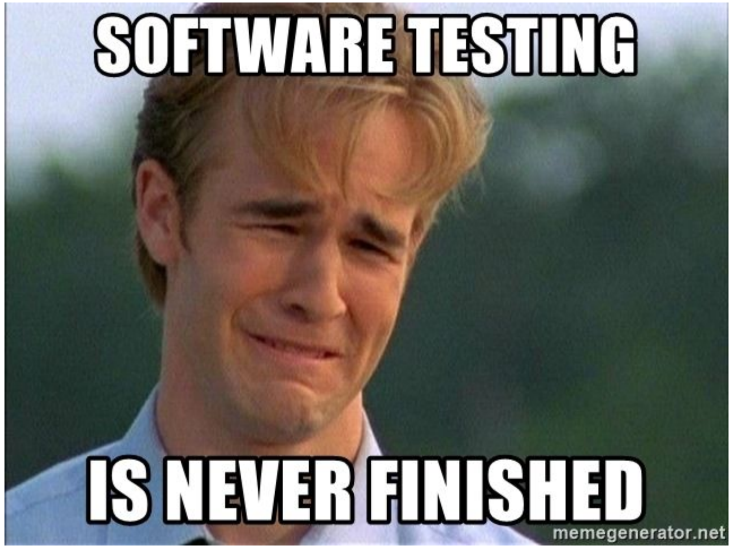
When to not use assertions

- For stuff that seems really obvious
  - It might be overkill to assert a type for the input of a factorial function
  - I really don’t need to assert that 5 is an integer…
- Don’t use asserts as a way to stop other people from using your functions in bad ways.
  - Don’t say in your docstring “throws an assertion error if given invalid input” because assert statements can be turned off.
  - For Python code used in most analytical contexts, using asserts in this way is fine, but you wouldn’t want to do this in production code.

Remember: Testing doesn’t prove correctness

It only will prove if there are bugs present.

If you choose good inputs for your tests and run them often, you can be more confident that you’ve caught bugs.



Testing Floats

Weird behavior...

Example:

```
def multiply_by_three(x):
     return x + x + x

f1 = multiply_by_three(0.1)
f2 = 0.3
assert(f1 == f2)
```

Will this cause an assertion error! Why?

Example:

```
def multiply_by_three(x):
    return x + x + x

f1 = multiply_by_three(0.1)
f2 = 0.3
# adding a debugging print statement
print(f1, f2)
```

Output: `0.30000000000000004 0.3`

The way Python represents floats causes things like this to happen

Example:

```
def multiply_by_three(x):
     return x + x + x

f1 = multiply_by_three(0.1)
f2 = 0.3

assert(math.isclose(f1, f2))
```

No error anymore! Might be useful on a future homework…

Real Life Examples

Homework 4

majority_count

```
def majority_count(labels):
    """Return the count of the majority labels in the label list

    Arguments:
        labels: a list of labels

    Returns: the count of the majority labels in the list
    """
```

```python
def majority_count(labels):
    """Return the count of the majority
      labels in the label list"""
```

majority_count tests

```python
# single
assert majority_count([0, 0, 0, 0, 0, 0]) == 6

# mixed
assert majority_count([0, 0, 1, 1, 0, 0]) == 4
assert majority_count([0, 2, 2, 2, 3, 3, 0, 1, 1, 0, 0]) == 4

# tied max count
assert majority_count([0, 2, 2, 2, 0, 2, 0, 0]) == 4
print("test_majority_count passed")
```

Do you see any potential issues with these tests?

```python
def majority_count(labels):
    """Return the count of the majority
      labels in the label list"""
```

Issues with majority_count tests

```python
# single
assert majority_count([0, 0, 0, 0, 0, 0]) == 6

# mixed
assert majority_count([0, 0, 1, 1, 0, 0]) == 4
assert majority_count([0, 2, 2, 2, 3, 3, 0, 1, 1, 0, 0]) == 4

# tied max count
assert majority_count([0, 2, 2, 2, 0, 2, 0, 0]) == 4
print("test_majority_count passed")
```

- The majority label is always 0
  - Code that was simply:
    `return labels.count(0)`
    Would pass these tests, despite being very wrong
- The majority label is always the last element in the list
  - Say your code creates a dictionary where the keys are the labels, and the values are the counts of the label. But instead of finding the largest count, you just returned the last count seen. This would pass all tests, despite being very wrong.

```python
def majority_count(labels):
    """Return the count of the majority
      labels in the label list"""
```

Issues with majority_count tests

```python
# single
assert majority_count([0, 0, 0, 0, 0, 0]) == 6

# mixed
assert majority_count([0, 0, 1, 1, 0, 0]) == 4
assert majority_count([0, 2, 2, 2, 3, 3, 0, 1, 1, 0, 0]) == 4

# tied max count
assert majority_count([0, 2, 2, 2, 0, 2, 0, 0]) == 4
print("test_majority_count passed")
```

- The majority label is always 0
  - Code that was simply:
    `return labels.count(0)`
    Would pass these tests, despite being very wrong
- The majority label is always the last element in the list
  - Say your code creates a dictionary where the keys are the labels, and the values are the counts of the label. But instead of finding the largest count, you just returned the last count seen. This would pass all tests, despite being very wrong.

What are some additional tests we could write?

```python
def majority_count(labels):
    """Return the count of the majority
      labels in the label list"""
```

Issues with majority_count tests

```python
# single
assert majority_count([0, 0, 0, 0, 0, 0]) == 6

# mixed
assert majority_count([0, 0, 1, 1, 0, 0]) == 4
assert majority_count([0, 2, 2, 2, 3, 3, 0, 1, 1, 0, 0]) == 4

# tied max count
assert majority_count([0, 2, 2, 2, 0, 2, 0, 0]) == 4


# the majority label is not the last element in the list
assert majority_count([0, 0, 1]) == 2

# the majority label is not 0
assert majority_count([1, 1, 0]) == 2
```

- The majority label is always 0
  - Code that was simply:
    `return labels.count(0)`
    Would pass these tests, despite being very wrong
- The majority label is always the last element in the list
  - Say your code creates a dictionary where the keys are the labels, and the values are the counts of the label. But instead of finding the largest count, you just returned the last count seen. This would pass all tests, despite being very wrong.

What are some additional tests we could write?

Homework Advice

- Don’t blindly trust the fact that all your tests pass
- Consider writing your own tests based on your own understanding
- Especially important for the last homework -- you’ll have to write all the tests yourself!

Section Handout

Let’s get comfortable writing tests!

1.

Consider the following function:

```
def max_even(lst):
    """
    Returns the maximum even valued integer in lst.
    Keyword arguments: lst -- a list of integers
    If lst has no maximum even value, returns None
    """
    # Implementation not shown
```

Write tests to test the following cases for possibilities of the argument’s, lst’s, value. Each test should be about one line long.

2

Write tests for the following function:

```
def multiply(a, b):
    """
    Computes the product of the numbers a and b
    """
```

3

Write tests for the following function:

```
def mode(lst):
    """
    Returns the mode, defined as the most common value, in lst
    Keyword arguments:
    lst -- a list of integers
    If len(lst) == 0, returns None
    If multiple modes are possible, returns one of the potential values.
    Which value returned is not explicitly defined.
    """
```