TESTING

Objectives

- Describe what tests are and why they are essential
- Explain what Test Driven Development is
- Test Python code using doctests
- Test Python code using assert
- Explain what unit testing is
- Write unit tests using the unittest module
- Remove code duplication using before and after hooks

Why test?

Why test?

- Reduce bugs in existing code
- Ensure bugs that are fixed stay fixed
- Ensure that new features don't break old ones
- Ensure that cleaning up code doesn't introduce new bugs
- Makes development more fun!

Test Driven Development

- Development begins by writing tests
- Once tests are written, write code to make tests pass
- Once tests pass, a feature is considered complete

Red, Green, Refactor

1. Red - Write a test that fails
2. Green - Write the minimal amount of code necessary to make the test pass
3. Refactor - Clean up the code, while ensuring that tests still pass

Assertions

Assertions

- We can make simple assertions with the `assert` keyword
- `assert` accepts an expression
- Returns `None` if the expression is truthy
- Raises an `AssertionError` if the expression is falsy
- Accepts an optional error message as a second argument

Assertions Example

```python
def add_positive_numbers(x, y):
    assert x > 0 and y > 0, "Both numbers must be positive!"
    return x + y

add_positive_numbers(1, 1) # 2
add_positive_numbers(1, -1) # AssertionError: Both numbers must be positive!
```

Assertions Warning

If a Python file is run with the -O flag,
assertions will not be evaluated!

```python
# Don't write code like this!

def do_something_bad(user):
    assert user.is_admin, "Only admins can do bad things!"
    destroy_a_bunch_of_stuff()
    return "Mua ha ha ha!"
```

doctests

doctests

- We can write tests for functions inside of the docstring
- Write code that looks like it's inside of a REPL

doctests Example

```python
def add(x, y):
    """add together x and y

    >>> add(1, 2)
    3

    >>> add(8, "hi")
    Traceback (most recent call last):
        ...
    TypeError: unsupported operand type(s) for +: 'int' and 'str'
    """
```

Run these tests with:

```
python3 -m doctest -v YOUR_FILE_NAME.py
```

Test should fail at first - remember "Red, Green, Refactor"

Issues with doctests

- Syntax is a little strange
- Clutters up our function code
- Lacks many features of larger testing tools
- Tests can be brittle

YOUR
TURN

Introduction to `unittest`

Unit testing

- Test smallest parts of an application in isolation (e.g. units)
- Good candidates for unit testing: individual classes, modules, or functions
- Bad candidates for unit testing: an entire application, dependencies across several classes or modules

`unittest`

- Python comes with a built-in module called `unittest`
- You can write unit tests encapsulated as classes that inherit from `unittest.TestCase`
- This inheritance gives you access to many assertion helpers that let you test the behavior of your functions
- You can run tests by calling `unittest.main()`

unittest Example

activities.py

```python
def eat(food, is_healthy):
    pass

def nap(num_hours):
    pass
```

tests.py

```python
import unittest
from activities import eat, nap

class ActivityTests(unittest.TestCase):
    pass

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

Commenting Tests

`tests.py`

```
class SomeTests(unittest.TestCase):
    def first_test(self):
        """testing a thing"""
        self.assertEqual(thing(), "something")

    def second_test(self):
        """testing another thing"""
        self.assertEqual(another_thing(), "something else")
```

To see comments, run

```
python NAME_OF_TEST_FILE.py -v
```

Types of Assertions

- `self.assertEqual(x, y)`
- `self.assertNotEqual(x, y)`
- `self.assertTrue(x)`
- `self.assertFalse(x)`
- `self.assertIsNone(x)`
- `self.assertIsNotNone(x)`
- `self.assertIn(x, y)`
- `self.assertNotIn(x, y)`
- ...and more!

Testing for Errors

```
class SomeTests(unittest.TestCase):
    def testing_for_error(self):
        """testing for an error"""
        with self.assertRaises(IndexError):
            l = [1,2,3]
            l[100]
```

Before and After Hooks

setUp and tearDown

- For larger applications, you may want similar application state before running tests
- setUp runs before each test method
- tearDown runs after each test method
- Common use cases: adding/removing data from a test database, creating instances of a class

Example

```
class SomeTests(unittest.TestCase):

    def setUp(self):
        # do setup here
        pass

    def test_first(self):
        # setUp runs before
        # tearDown runs after
        pass

    def test_second(self):
        # setUp runs before
        # tearDown runs after
        pass

    def tearDown(self):
        # do teardown here
        pass
```

Recap

- Tests help streamline development and reduce bugs
- You can start with tests if doing TDD / Red, Green, Refactor
- You can perform simple checks with `assert`
- You can test with doctests, but typically shouldn't
- `unittest` is a feature-rich, OOP style testing library in Python
- To reduce code duplication in tests, use before/after hooks!

YOUR
TURN