# Unit Testing - pytest

## CS 2ME3/SE 2AA4

Zichen Jiang
Owen Huyn

Department of Computing and Software
McMaster University

February 9, 2018

# Outline

# What is Unit Testing

- Unit testing verifies that individual units of code (usually functions) work as intended
- Designed to be **simple**, easy to write and run
- You can test both from a blackbox perspective and a whitebox perspective

# Who writes unit tests

- Developers should test their own code!
- The person who wrote the code usually has the best understanding of what their code does

# Why Unit Test?

- You can catch bugs much earlier
- Provides documentation on a specific function
- Helps developer improve the implementation/design of a function
- Every good developer should be a good tester too!
  No one likes to work with someone who doesn't verify/test if their code works.

# What is pytest

- pytest is a framework that makes it easy to write small tests
- Similar to JUnit (Java), CppUnit (C++)
- Knowledge is transferable to another xUnit framework regardless of language

# Getting Started

- Installing pytest is very simple, simply run the following command in your command line
- `pip install -U pytest`
- For more information, visit pytest Installation and Getting Started

# Demo

- Let's get started, I encourage everyone to pull out their laptops and follow along.
- We will test A1 from last year, which was introduced in the past tutorials
- Don't be afraid to ask any questions!

# Create our first unit test file

- To start, create a new Python file called test_circles.py in the same directory of our file that we want to test
- You can do this from the command line or any text editor of your choice. Don't be afraid to ask any questions!

# Create our first test template

- To start with our unit test, follow this template:

```
import pytest

class TestCircles:
```

1. Import the pytest library
2. Write a unit testing class starting with the word `Test`

# What is Assert?

- Wikipedia: "... an assertion is a statement that a predicate (Boolean-valued function, i.e. a true-false expression) is expected to always be true at that point in the code. If an assertion evaluates to false at run time, an assertion failure results, which typically causes the program to crash, or to throw an assertion exception."
- Basically, if whatever follows assert is true, it will continue. Otherwise the test will fail and stop running.

# Example of Assert?

- To assert something to be true, you can write

```
assert <true statement>
```

- To assert something to be false, you can write

```
assert not <false statement>
```

# Writing our first test

```
import pytest
from CircleADT import *

class TestCircles:

  def test_xcoord_are_equal(self):
    circle = CircleT(1,2,3)
    assert circle.xcoord() == 1

  def test_xcoord_are_not_equal(self):
    circle = CircleT(1,2,3)
    assert not (circle.xcoord() == 2)
    # or 'assert circle.xcoord() != 2'
```

# Running our first test

- Run this command in your command prompt inside the folder of your test file: `pytest test_circles.py`
- To run all tests within the directory, just run `pytest`. It will search for all files starting with "test_" and run all methods starting with "test_"

```
Alics-MacBook-Pro:src Alic$ pytest
============================= test session starts ==============================
platform darwin -- Python 3.4.3, pytest-3.3.2, py-1.5.2, pluggy-0.6.0
rootdir: /Users/Alic/git/se2aa4_cs2me3/Tutorials/T05-PyUnit/src, inifile:
plugins: cov-2.5.1
collected 5 items

test_circles.py ..                                                       [ 40%]
test_statistics.py ...                                                   [100%]

========================== 5 passed in 0.18 seconds ===========================
Alics-MacBook-Pro:src Alic$
```

# What if a test failed?

- Say we have the following code:

```
import pytest
from CircleADT import *

class TestCircles:

  def test_xcoord_are_equal(self):
    circle = CircleT(1,2,3)
    assert circle.xcoord() == 1

  def test_xcoord_are_not_equal(self):
    circle = CircleT(1,2,3)
    assert circle.xcoord() != 1
```

# What if a test failed?

```
Alics-MacBook-Pro:src Alic$ pytest test_circles.py
============================= test session starts ==============================
platform darwin -- Python 3.4.3, pytest-3.3.2, py-1.5.2, pluggy-0.6.0
rootdir: /Users/Alic/git/se2aa4_cs2me3/Tutorials/T05-PyUnit/src, inifile:
plugins: cov-2.5.1
collected 2 items

test_circles.py .F                                                       [100%]

=================================== FAILURES ===================================
_____________________ TestCircles.test_xcoord_are_not_equal _____________________

self = <test_circles.TestCircles object at 0x1039b3860>

    def test_xcoord_are_not_equal(self):
>       assert self.circle.xcoord() != 1
E       assert 1 != 1
E        +  where 1 = <bound method CircleT.xcoord of <CircleADT.CircleT object a
t 0x1039b3898>>()
E        +    where <bound method CircleT.xcoord of <CircleADT.CircleT object at
0x1039b3898>> = <CircleADT.CircleT object at 0x1039b3898>.xcoord
E        +      where <CircleADT.CircleT object at 0x1039b3898> = <test_circles.T
estCircles object at 0x1039b3860>.circle

test_circles.py:28: AssertionError
==================== 1 failed, 1 passed in 0.08 seconds ====================
```

# Floating Point assertions

- `approx(expected, rel=None, abs=None, nan_ok=False)`
  - `rel`: relative tolerance
  - `abs`: absolute tolerance
  - `nan_ok`: facilitates comparing arrays that use NaN to mean "no data"
- By default, `approx` considers numbers within a relative tolerance of 1e-6 and absolute tolerance of 1e-12 of its expected value to be equal.

```
>>> 1.0001 == approx(1)
False
>>> 1.0001 == approx(1, rel=1e-3)
True
>>> 1.0001 == approx(1, abs=1e-3)
True
```

# Redundant code in tests

```
import pytest
from CircleADT import *

class TestCircles:

  def test_xcoord_are_equal(self):
    circle = CircleT(1,2,3)
    assert circle.xcoord() == 1

  def test_xcoord_are_not_equal(self):
    circle = CircleT(1,2,3)
    assert circle.xcoord() = 1
```

# Cleaned up code

```
import pytest
from CircleADT import *

class TestCircles:

  def setup_method(self, method):
    self.circle = CircleT(1,2,3)

  def teardown_method(self, method):
    self.circle = None

  def test_xcoord_are_equal(self):
    assert self.circle.xcoord() == 1

  def test_xcoord_are_not_equal(self):
    assert self.circle.xcoord() = 2
```

# Setup and Teardown

- `setup_class(cls)`
  - setup any state specific to the execution of the given class
- `teardown_class(cls)`
  - teardown any state that was previously setup with a call to `setup_class`.
- `setup_method(self, method)`
  - setup any state tied to the execution of the given method in a class. `setup_method` is invoked for every test method of a class.
- `teardown_method(self, method)`
  - teardown any state that was previously setup with a `setup_method` call.

# Asserting about exceptions

- It is very important in unit testing to check for edge cases and behaviour in the case of unexpected input.
- If a function raises an exception in some cases, you should include those cases in your unit testing as well.
- You can do so by using a context manager called pytest.raises.

```
import pytest

def test_zero_division():
with pytest.raises(ZeroDivisionError):
1 / 0
```

# Asserting about exceptions

- If an exception is not raised in a pytest.raises block, the test will fail.

```
import pytest

def test_zero_division():
  with pytest.raises(ZeroDivisionError):
    2 / 1
```

- If you run this test, it will give the result:

```
================================== FAILURES ===================================
_____________________________ test_zero_division ______________________________

    def test_zero_division():
        with pytest.raises(ZeroDivisionError):
>               2/1
E     Failed: DID NOT RAISE <class 'ZeroDivisionError'>

test_exception_not_raised.py:5: Failed
========================== 1 failed in 0.06 seconds ===========================
```

# How much should I test?

- Test all requirements in each function
- Cover edge cases that may cause unintended consequences
- Have an acceptable amount of code coverage
- Code coverage will be covered in more detail in future lectures

# Code Coverage

- Function coverage: has each function (or subroutine) in the program been called?
- Statement coverage: has each statement in the program been executed?

# Code Coverage

- Branch coverage: has each branch of each control structure (such as in if and case statements) been executed?
  - For example, given an if statement, have both the true and false branches been executed?
  - Another way of saying this is, has every edge in the program been executed?
- Condition coverage (or predicate coverage): has each Boolean sub-expression evaluated both to true and false?

# Pytest plugin for measuring coverage

- https://pypi.python.org/pypi/pytest-cov
- Install with pip: `pip install pytest-cov`

# Running the coverage plugin

- In the source code directory, run `pytest --cov`

```
Alics-MacBook-Pro:src Alic$ pytest --cov
============================= test session starts ==============================
platform darwin -- Python 3.4.3, pytest-3.3.2, py-1.5.2, pluggy-0.6.0
rootdir: /Users/Alic/git/se2aa4_cs2me3/Tutorials/T05-PyUnit/src, inifile:
plugins: cov-2.5.1
collected 5 items

test_circles.py ..                                                       [ 40%]
test_statistics.py ...                                                   [100%]

---------- coverage: platform darwin, python 3.4.3-final-0 -----------
Name                 Stmts   Miss  Cover
----------------------------------------
CircleADT.py            35     17    51%
Statistics.py           22      0   100%
test_circles.py         13      0   100%
test_statistics.py      18      0   100%
----------------------------------------
TOTAL                   88     17    81%


========================== 5 passed in 0.43 seconds ===========================
Alics-MacBook-Pro:src Alic$
```

- We can see that `CircleADT.py` has onlyl 51% coverage. This is because only the methods `xcoord` and `ycoord` were tested

# Exercise

- The rest of the methods are left for you as practices
- You can Refer to testStatistics.py for a more complete breakdown on how to test complicated functions

# References


- https://docs.pytest.org/en/latest/xunit_setup.html
- https://en.wikipedia.org/wiki/Code_coverage
- http://pytest.readthedocs.io/en/reorganize-docs/new-docs/user/pytest_raises.html