COMP20230: Data Structures & Algorithms
Lecture 8: Testing and OOP

Dr Andrew Hines

Office: E3.13 Science East
School of Computer Science
University College Dublin

andrew.hines@ucd.ie

Outline: Testing strategies and OOP

Testing? Is this not Algorithms and Data Structures

Yes, but you need to be able to implement your algorithms and data structures in code (specifically Python) and to test if they work.

Recap OOP Concepts to reinforce ADT concepts:
Abstract Data Types – Data structures exist in functional and OOP.
APIs and Classes → Encapsulation → ADT
Class attributes (data structures) and methods (algorithms)

Take home message

Test driven development makes for better design, development and testing
Better code, and better algorithm implementations
But lots of testing doesn't make up for bad design!

Testing our Algorithms and Data structures

Attributes of a good data structure

Clarity, Robustness, Correctness, Maintainability (Security, Speed, Efficiency)

Quality

- Testing is good but it is not everything[1]
- Specifications need to be right too
- Testing: Unit Testing in Python, Regression Testing, Automated Testing, Assessment Criteria, Code Coverage

[1] `http://se.ethz.ch/~meyer/publications/testing/principles.pdf`

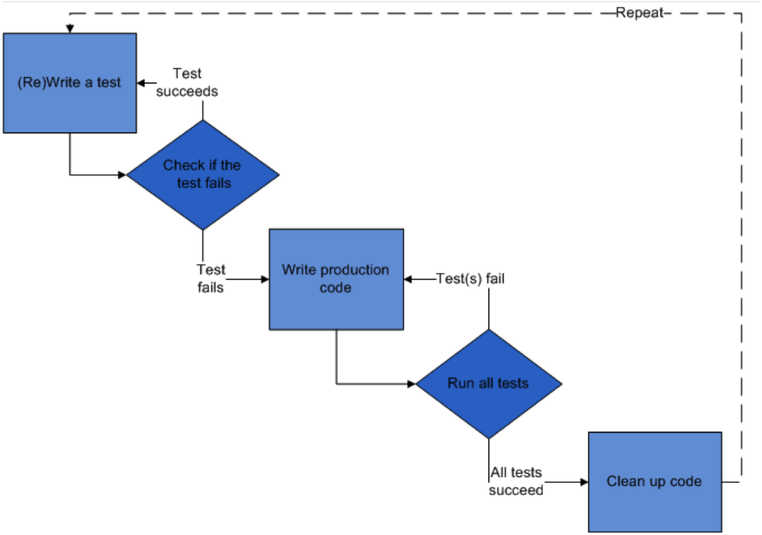
Testing our Algorithms and Data structures

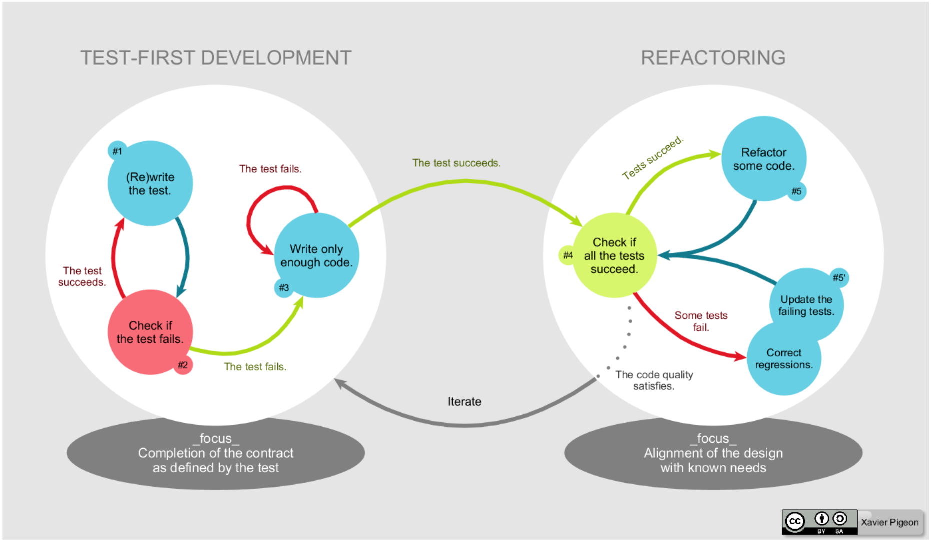
Testing Strategies

- Regression Testing: Any failed execution must yield a test case, to remain a permanent part of the project's test suite.
- Unit Testing: Test the expected and failed executions of components algorithms and APIs
- Oracles: assessment criteria determining if a test has unambiguously passes
- Automated Testing (needed for oracles)
- Code Coverage: Are lines of code executed by the tests?

Evaluate: any testing strategy, however attractive in principle, through objective assessment using explicit criteria in a reproducible testing process.
Assess: A testing strategy's most important property is the number of faults it uncovers as a function of time.

Test Driven Development





"TDD Global Lifecycle" by Xarawn - Own work. Licensed under CC BY-SA 4.0 via Commons - https://commons.wikimedia.org/wiki/File:TDD_Global_Lifecycle.png#/media/File:TDD_Global_Lifecycle.png

Fear!

Fear

- Makes you tentative
- Makes you grumpy
- Makes you want to communicate less
- Makes you shy from feedback

Fear!

Fear

- Makes you tentative
- Makes you grumpy
- Makes you want to communicate less
- Makes you shy from feedback

TDD can fight fear!

- Instead of being tentative, begin learning concretely as quickly as possible.
- Instead of clamming up, communicate more clearly.
- Instead of avoiding feedback, search out helpful, concrete feedback.
- (You have to work on grumpiness on your own!)

From: Test Driven Development: By Example, Kent Beck, Addison-Wesley Longman, 2002, ISBN 0-321-14653-0, ISBN 978-0321146533

Leap of Faith

Bridge the gap between the decision and feedback during programming.

Will it work... will it... will it... no! Why not?!

- How many lines of code do you write before hitting run?
- How many times do you hit run before you are confident the program works as expected?

https://upload.wikimedia.org/wikipedia/commons/7/70/Mind-the-gap.jpg

Precise language and communication reduces mistakes

IEEE standard terminology

An unsatisfactory program execution is a *failure*,
pointing to a *fault* in the program,
itself the result of a *mistake* in the programmer's thinking.
The informal term *bug* can refer to any of these phenomena.

Precise language and communication reduces mistakes

IEEE standard terminology

An unsatisfactory program execution is a *failure*,
pointing to a *fault* in the program,
itself the result of a *mistake* in the programmer's thinking.
The informal term *bug* can refer to any of these phenomena.

In order to minimise mistakes

we will shortly recap OOP terminology!

Unit Testing

Why unit test?

- Verify that the program is doing what we expected
- Pinpoint "bugs" (fix mistake by identifying fault and removing failure!), makes debugging easier
- It's a habit, start it early!

1. Define a test case (what you actually want to test)
2. Confirm that the actual output matches the expected output
3. In unit testing frameworks you will often assert if something is True or False (Note: you can do more than this)
4. You can check that the actual behaviour matches the expected behaviour e.g. This could be an operation of a specific ADT for example

Python unit testing framework

- We can use Python's `unittest` module as a unit testing framework: Documentation at: `https://docs.python.org/3.4/library/unittest.html`
- Within the this module, we have the `TestCase` class which we need to subclass (this means that we **inherit** `TestCase`'s methods).
- Here, we import the `unittest` module and the class we want to test

```
import unittest
from examples.triangle import Triangle

class TestTriangle(unittest.TestCase):
```

Creating a test

Test naming convention

Start method name with the word ‘test’.
This is how we let the tester knows that these methods are unit tests.

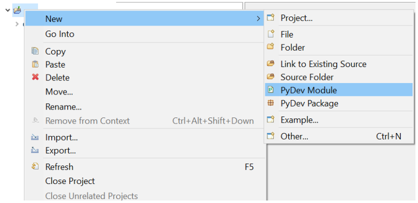
```
def test_invalid1(self):
    t = Triangle(1, 2, 4)
    self.assertTrue(t.classify() == "INVALID")
```

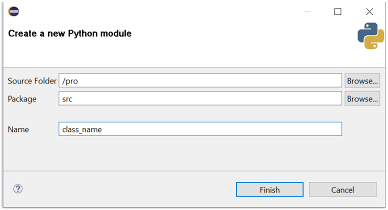
Use `TestCase`’s methods like `assertTrue()` and `assertFalse()` to test output.

Organising your tests

Keeping a structure to your code

Organise your tests: put them in a python module within your project folder
This is easily done in IDEs like Eclipse or PyCharm*




Create a new Python module

Source Folder: /pro
Package: src
Name: class_name

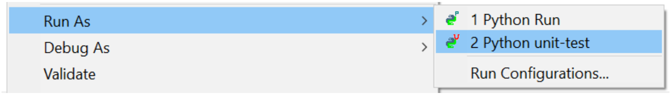
*In Pycharm you can do this by creating a new package.

Running Tests

In order to get the unittest framework to automatically run your tests, add a line at the bottom of the module that tells python to call it automatically if the `<modulename.py>` file is called.

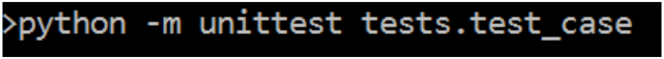
```
if __name__ == "__main__":
    unittest.main()
```

In Eclipse, you can run tests by right clicking the file and selecting 'Python unit-test':



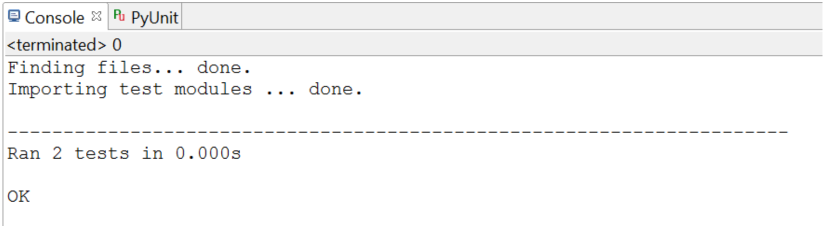
Running from the command line

Navigate into the project folder: `tests` in the example below is the Python module/package and `test_case` is the class within this module



```
>python -m unittest tests.test_case
```

Whether you run from the command line or in an IDE you'll see your test results:

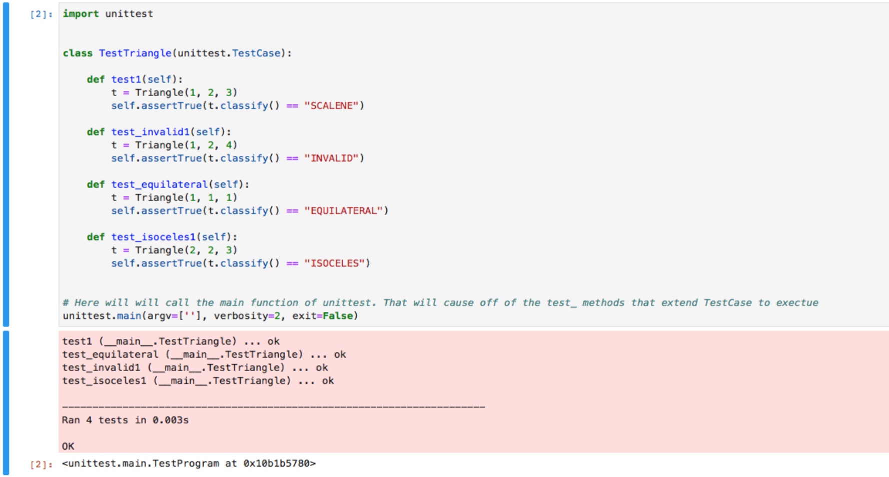


```
Finding files... done.
Importing test modules ... done.

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Running from Jupyter Lab/Notebook



```
[2]: import unittest


     class TestTriangle(unittest.TestCase):

         def test1(self):
             t = Triangle(1, 2, 3)
             self.assertTrue(t.classify() == "SCALENE")

         def test_invalid1(self):
             t = Triangle(1, 2, 4)
             self.assertTrue(t.classify() == "INVALID")

         def test_equilateral(self):
             t = Triangle(1, 1, 1)
             self.assertTrue(t.classify() == "EQUILATERAL")

         def test_isoceles1(self):
             t = Triangle(2, 2, 3)
             self.assertTrue(t.classify() == "ISOCELES")


     # Here will will call the main function of unittest. That will cause off of the test_ methods that extend TestCase to exectue
     unittest.main(argv=[''], verbosity=2, exit=False)
```

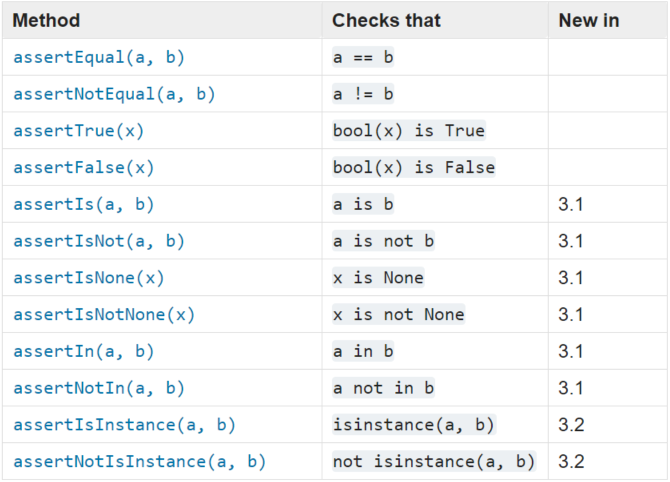
```
test1 (__main__.TestTriangle) ... ok
test_equilateral (__main__.TestTriangle) ... ok
test_invalid1 (__main__.TestTriangle) ... ok
test_isoceles1 (__main__.TestTriangle) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.003s

OK
```

```
[2]: <unittest.main.TestProgram at 0x10b1b5780>
```

Unit Test Framework Assert Methods



Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

Online Resources

Testing

http://www.diveintopython3.net/unit-testing.html
https://docs.python.org/3/library/unittest.html
http://docs.python-guide.org/en/latest/writing/tests/

Good Introduction to TDD

Test Driven Development: By Example, Kent Beck, Addison-Wesley Longman, 2002, ISBN 0-321-14653-0, ISBN 978-0321146533

Debugging in IDEs

Aside

Working in Notebooks means we don't get the opportunity to debug in an IDE. Understanding breakpoints and watches are key skills in developing your ability to analyse and debug algorithm implementations and software systems

PyCharm IDE Docs and Tutorials

https://www.jetbrains.com/help/pycharm/2016.3/debugging.html
https://www.jetbrains.com/help/pycharm/2016.3/debug-tool-window.html

OOP Revision: Terminology

Class

A class is a template that defines objects of a new data type by specifying their state and behaviour.

Example

A `PlayingCard` class might define card objects for a game program.

Terminology

Object

An object is a specific instance of a class, with its own particular state. A program may create as many instances of a class as it needs.

Example

A card game might create 52 PlayingCard objects to represent a deck.

Terminology

State

Objects have attributes or characteristics that are important to track for the purposes of a particular application.
The current values of those attributes constitute the object's state.
Object state is stored in fields, sometimes more specifically referred to as instance fields.

Example

A `PlayingCard` might have fields to store its rank and suit.

Terminology

Behaviour

Objects will also have behaviours that need to be modelled by an application. Object behaviours are described by methods.

Example

A `CardDeck` class might define a `shuffle()` method.

Terminology

Mutators and Accessors

Methods are called *mutators* if they change the state of the object or *accessors* if they only return information about the state.

Examples

A `CardDeck` class might define a `shuffle()` method as a *mutator*.
A `PlayingCard` class might define a `getSuit()` method as a *accessor*.

Terminology

Constructor

Constructors are called to create new objects. Each class will have its own constructor.

Terminology

Encapsulation

- Providing an interface for software that simplifies its usage
- Encapsulation is the packing of data and functions into a single component
- hide data member and member function implementations

Terminology

Encapsulation

- Providing an interface for software that simplifies its usage
- Encapsulation is the packing of data and functions into a single component
- hide data member and member function implementations

WHY?

Ease of use and robustness

Terminology

Instantiation

Instantiation creating an instance of a class
The representation of (an abstraction) by a concrete example

An object is a specific version of a class that can have unique attributes.
It will have its on storage place in the computer memory
e.g. if there was a `class` called Human, “Tom” could be an instance of human

Creating Classes

Everything is a class

All data types in Python are object data types and so are defined by classes.

What is different now is that we are writing classes to create new data types of our own.

In Summary...

The idea behind OOP
The terminology to describe it
Breaking a class down by Nouns, Verbs, Adjectives