UNIVERSITY OF Hull

Wrestling with Python
Unit testing

Warren Viant

Assessment criteria – OCR - 2015

- Programming Techniques (12 marks)
  - There is an attempt to solve all of the tasks using most of the techniques listed.
  - The techniques are used appropriately in all cases giving an efficient, working solution for all parts of the problem.
- Design (9 marks)
  - There is a detailed analysis of what is required for these tasks justifying their approach to the solution.
  - There will be a full set of detailed algorithms representing a solution to each part of the problem.
  - There is **detailed discussion of testing and success criteria**.
  - The variables and structures are identified together with any validation required.
- Development (9 marks)
  - There is detailed evidence showing development of the solution **with evidence of systematic testing during development** to show that all parts work as required.
  - The code is well organised with meaningful variable names and detailed annotation indicating the function of each section.
- Testing (9 marks)
  - **The test plan covers all major success criteria** for the original problem with evidence to show how each of these criteria have been met, or if they have not been met, how the issue might be resolved.
  - There is a **full evaluation of the final solution against the success criteria**.
  - A high level of written communication will be obvious throughout the task and specialist terms/technology with accurate use of spelling will have been used.
  - Grammar and punctuation are used correctly and information is presented in a coherent and structured format.

How to test your code?

- Traditional approach
  - Plan a series of tests on paper
  - Manually conduct the tests
  - If there is a failure, fix the code and repeat the tests

- Do your pupils enjoy testing?

Test driven development (TDD)

- Before writing any Python code you first write an automated test. While writing the test you can focus on the correct input and output for your code. The advantage is that at this stage you are not concerned with how you implement your code.
- Run your test and your test should fail. This is ok, as you have yet to implement your algorithm.
- Begin programming your algorithm.
- Run your test again, if it fails it means you have more work to do.
- Once the code passes the test, move onto the next algorithm.

Planning your test

- Think about what the function should do
- What inputs it will need
- What outputs are expected

Example Python unit test

- Python provides an automated testing facility that is linked to PyCharm

```
1. import unittest
2. from Task_1 import *

3. class TestSequenceFunctions(unittest.TestCase):

4.     def test_rollDie(self):
5.         maxNumOfSides = 20
6.         for die in range(2, maxNumOfSides+1):
7.             value = rollDie(die)
8.             self.assertTrue(value >= 1)
9.             self.assertTrue(value <= maxNumOfSides)
```

Example Python unit test

- Python provides an automated testing facility that is linked to PyCharm

```
1. import unittest
2. from Task_1 import *

3. class TestSequenceFunctions(unittest.TestCase):

4.     def test_rollDie(self):
5.         maxNumOfSides = 20
6.         for die in range(2, maxNumOfSides+1):
7.             value = rollDie(die)
8.             self.assertTrue(value >= 1)
9.             self.assertTrue(value <= maxNumOfSides)
```

Example Python unit test

- Python provides an automated testing facility that is linked to PyCharm

```
1. import unittest
2. from Task_1 import *

3. class TestSequenceFunctions(unittest.TestCase):

4.     def test_rollDie(self):
5.         maxNumOfSides = 20
6.         for die in range(2, maxNumOfSides+1):
7.             value = rollDie(die)
8.             self.assertTrue(value >= 1)
9.             self.assertTrue(value <= maxNumOfSides)
```

Example Python unit test

- Python provides an automated testing facility that is linked to PyCharm

```
1. import unittest
2. from Task_1 import *

3. class TestSequenceFunctions(unittest.TestCase):

4.     def test_rollDie(self):
5.         maxNumOfSides = 20
6.         for die in range(2, maxNumOfSides+1):
7.             value = rollDie(die)
8.             self.assertTrue(value >= 1)
9.             self.assertTrue(value <= maxNumOfSides)
```

Example Python unit test

- Python provides an automated testing facility that is linked to PyCharm

```
1. import unittest
2. from Task_1 import *

3. class TestSequenceFunctions(unittest.TestCase):

4.     def test_rollDie(self):
5.         maxNumOfSides = 20
6.         for die in range(2, maxNumOfSides+1):
7.             value = rollDie(die)
8.             self.assertTrue(value >= 1)
9.             self.assertTrue(value <= maxNumOfSides)
```

Steps to create a unit test

1. Create a new .py file for your test code
2. Import the Python unittest library
   - import unittest
3. Import your code to test
   - from <file name> import *
4. Create a class for your tests by inheriting from unittest.TestCase
   - class <class name> (unittest.TestCase):
5. Add one or more new test methods
   - def <method name>(self):
6. Add assert statements to your method to notify the system of an error
   - self.assertEqual(<variable 1> , <variable 2>)
   - self.assertTrue(<condition>)
7. Run the test

A sample of the assert methods available in Python's unittest

- If an *assert* method returns false, an error is logged and displayed in PyCharm

Method	Checks that
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b

- Documentation https://docs.python.org/3/library/unittest.html

Task 1 Simulating a dice (OCR 2014)

- A game uses dice with 4, 6 and 12 sides to determine various outcomes.
- Design, code and test a program that will simulate throwing dice with these numbers of sides.
- The user should be able to input which dice is being thrown, eg 4, 6 or 12.
- The program should output the dice chosen and the score e.g. “6 sided dice thrown, score 4”
- The user should be able to repeat this process as many times as required.

Process

1. Break down the problem into sub-problems
2. Consider writing a function for each sub-problem
   - What name best describes the function?
   - What task does the function perform?
   - What data is input?
   - What data is output?
3. Consider the test case for each function
   - What input data will break the function?
   - What is the correct output for each valid input?
4. Implement the functions and their unit tests