# TESTING

DR IAN CORNELIUS

# HELLO

- Learning Objectives
  1. Understand the concept of testing and how to test your code
  2. Demonstrate your knowledge of testing your code

# INTRODUCTION TO TESTING

- You may find yourself already having tested your code, this is **exploratory testing**
  - running your application for the first time and checking the features
- This form of testing is typically done without a plan
- No matter how well your application has been designed and coded, there will be some defects
- Testing is concerned with running your application with the intent of finding faults
- A successful test is one deemed to have found errors, not one that does not find any errors

# MANUAL OR AUTOMATED TESTING?

## 1. MANUAL TESTING

- Make a list of the following:
  - all features the application has
  - the different types of input accepted
  - any expected results
- Everytime a change is made to your code, you can go through the list
- Fairly tedious, and not much fun

## 2. AUTOMATED TESTING

- Execution of a test plan consisting of:
  - parts of the application you want to test
  - the order in which they are to be tested
  - any expected responses from functions
- The execution is performed by a script and not by yourself
- Python has a collection of tools and libraries to assist in automated testing
  - i.e. `pytest` and `unittest`

# INTEGRATION TESTING

- Integration tests look at the following:
  - interfaces between components
  - interactions between various parts of the system
  - file systems and hardware or interfaces between these systems
- This sort of testing is often performed after unit testing (more on that later)
- An integration test will interaction between two components and not the individual component functionality
  - think of it as if you are testing how a class interacts with another class
- You can consider performance testing to also be a part of this type of testing

# APPROACHES TO INTEGRATION TESTING (1)

- There are two approaches to integration testing:
  1. Big Bang
  2. Incremental
     - Top-Down
     - Bottom-Up
     - Sandwich

# APPROACHES TO INTEGRATION TESTING (2)

## BIG BANG

- All components and modules are integrated at once
- The unionising of different modules is then tested as a whole entity
- This approach will save time on testing and execution of the tests
- Test cases and their outcomes must be recorded correctly to ensure a robust test suite is performed
- **Advantages**:
  - the whole system is tested and requires minor planning
  - consists of completed and checked modules (unit testing)
  - often has no demand for urgent build fixings
- **Disadvantages**:
  - hard for modules and components to be separated if a bug has been detected
  - has a high risk to miss crucial issues when testing the whole system
  - failures often occur more frequently due to the simultaneous checking of numerous modules
  - one mistake can influence the results of the whole testing

# APPROACHES TO INTEGRATION TESTING (2)

## INCREMENTAL I

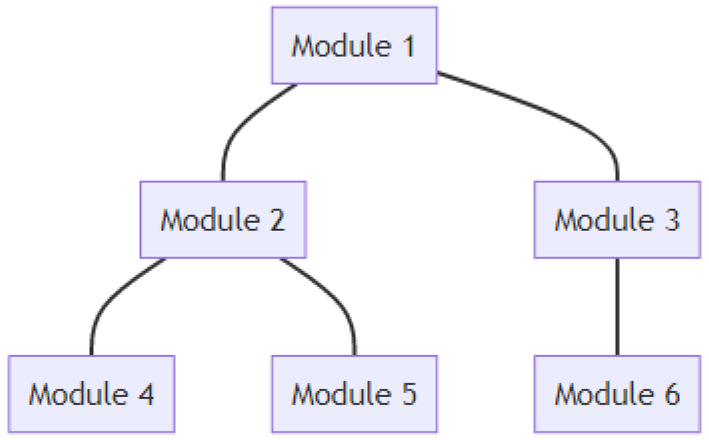
- Each element of the system is tested individually using unit tests
- Modules are then integrated incrementally and tested to ensure they interact correctly
- Primary focus of this test is to ensure that the interface and integrated links between modules work correctly
- The process is repeated until modules are combined and tested successfully
- Approaches towards this type of testing are:
  - Top-Down
  - Bottom-Up
  - Sandwich

# APPROACHES TO INTEGRATION TESTING (3)

## INCREMENTAL II

### TOP-DOWN

- Testing starts at the top and works towards the bottom
  - i.e. start with the central module to a sub-module
- **Advantages**:
  - provides early exposure to defects in the architecture
  - outlines the working of an application as a whole at an early stage
- **Disadvantages**:
  - important modules are tested later on in the cycle
  - can be quite challenging to write the test condition

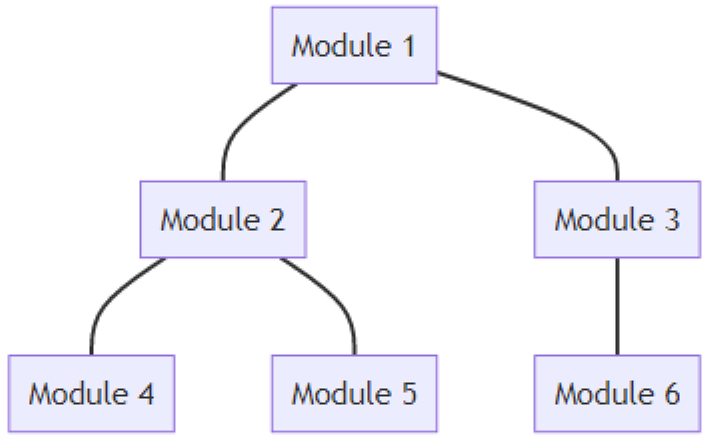


# APPROACHES TO INTEGRATION TESTING (4)

## INCREMENTAL III

### BOTTOM-UP

- Testing starts at the bottom and works towards the top
  - i.e. modules on the bottom layer are integrated and tested first, sequentially adding modules as integration moves up
- **Advantages**:
  - easier to create test-conditions
  - testing of critical modules' comes at an early stage, helps in an early discovery of errors
  - interface defects are detected at an earlier stage
- **Disadvantages**:
  - design defects are caught at a later stage
  - there is no working application until the last module is built



# APPROACHES TO INTEGRATION TESTING (5)

## INCREMENTAL IV

### SANDWICH

- Considered to be a hybrid of top-down and bottom-up incremental testing
- Middle layers are identified and a bottom-up and top-down testing approach is applied
  - the chosen middle layer is determined heuristically, i.e. selecting a layer with minial use of stubs and drivers
- **Advantages**:
  - beneficial for larger projects that has subprojects
  - top-down and bottom-up testing are run simultaneously
- **Disadvantages**:
  - before unification of modules, subsystems and interfaces are not tested thoroughly
  - not advised for systems that are highly inter-dependent with each other

# PERFORMING AN INTEGRATION TEST

- Performing an integration test can be done by following the collection of steps below:
    1. Prepare the integration test plan
    2. Design the test scenarios, cases and scripts
    3. Execute the test cases and follow-up with a report on the defects
    4. Tracking and re-testing of the defects
    5. Repeat steps three and four

# UNIT TESTING

- Unit testing looks at the individual units/components of an application
- The purpose is to validate each unit of an application performs correctly
- Mainly concerned with the following:
  - highlight the working and failing parts of an application
  - checking the input values and accuracy of the output data
  - optimisation of algorithms and performance
- **Advantages**:
  - each part of an application is tested individually
  - all components of an application is tested at least once
  - errors can be picked up earlier, and thus resolved earlier
  - the scope of testing is smaller, and thus easier to fix the errors

# PERFORMING A UNIT TEST

- Performing a unit test can be done by following the collection of steps below:
    1. Keep the unit tests small and fast
    2. Automate the tests to reduce turn-around
    3. Ensure the tests are simple to run
    4. Measure the outcome of the tests
    5. Fix any tests that fail immediately
    6. Keep testing at a unit level
    7. Name the tests appropriately
    8. Cover the boundary cases
    9. Provide a method of randomly generating data

# EXAMPLE OF A SIMPLE TEST IN PYTHON

- Unit test for checking the `sum()` function would require checking the output of `sum()` against a known output
  - i.e. check that the sum of numbers 4, 5 and 6 is equal to 15

```
assert sum([4, 5, 6]) == 15, "Should be 15"
```

- The above code will not display anything, as it satisfies to be `True`
- However, if we change the input for `sum()` to `[2, 3, 4]` we get a different result

```
assert sum([2, 3, 4]) == 15, "Should be 15"
```

- An `AssertionError` is thrown with the message `"Should be 15"`
- You can put this code into a Python file called `test_sum.py` and this will become a **test case**

# UNIT TESTING IN PYTHON

- The `unittest` module contains both a testing framework and test runner
- However, there are some important requirements when writing and executing unit tests:
  - tests are put into classes as methods
  - a series of special assertion functions are used instead of the built-in `assert` statement

# ASSERTION FUNCTIONS

| Method | Equivalent | Reverse |
|---|---|---|
| `assertEquals(a, b)` | a == b | `assertNotEqual(a, b)` |
| `assertTrue(x)` | bool(x) is True | n/a |
| `assertFalse(x)` | bool(x) is False | n/a |
| `assertIs(a, b)` | a is b | `assertIsNot()` |
| `assertIsNone(x)` | x is None | `assertIsNotNone()` |
| `assertIn(a, b)` | a in b | `assertNotIn()` |
| `assertIsInstance(a, b)` | isInstance(a, b) | `assertNotIsInstance()` |

- Further Reading:
    - Unit Testing in Python

# HOW TO CREATE A UNIT TEST

- You will create test methods to test each function in your application
  - it is best to prefix these test methods with `test_` followed by the name of the function you are testing

```
import unittest
class TestCases(unittest.TestCase):
    def test_sum(self):
        self.assertEqual(sum([4, 5, 6]), 15, 'Should be 15')
```

```
unittest.main()
```

- **Note**, that in this example I am using an in-built Python method
  - if you are using your own method from a different class/file you need to import it

# STRUCTURING A UNIT TEST

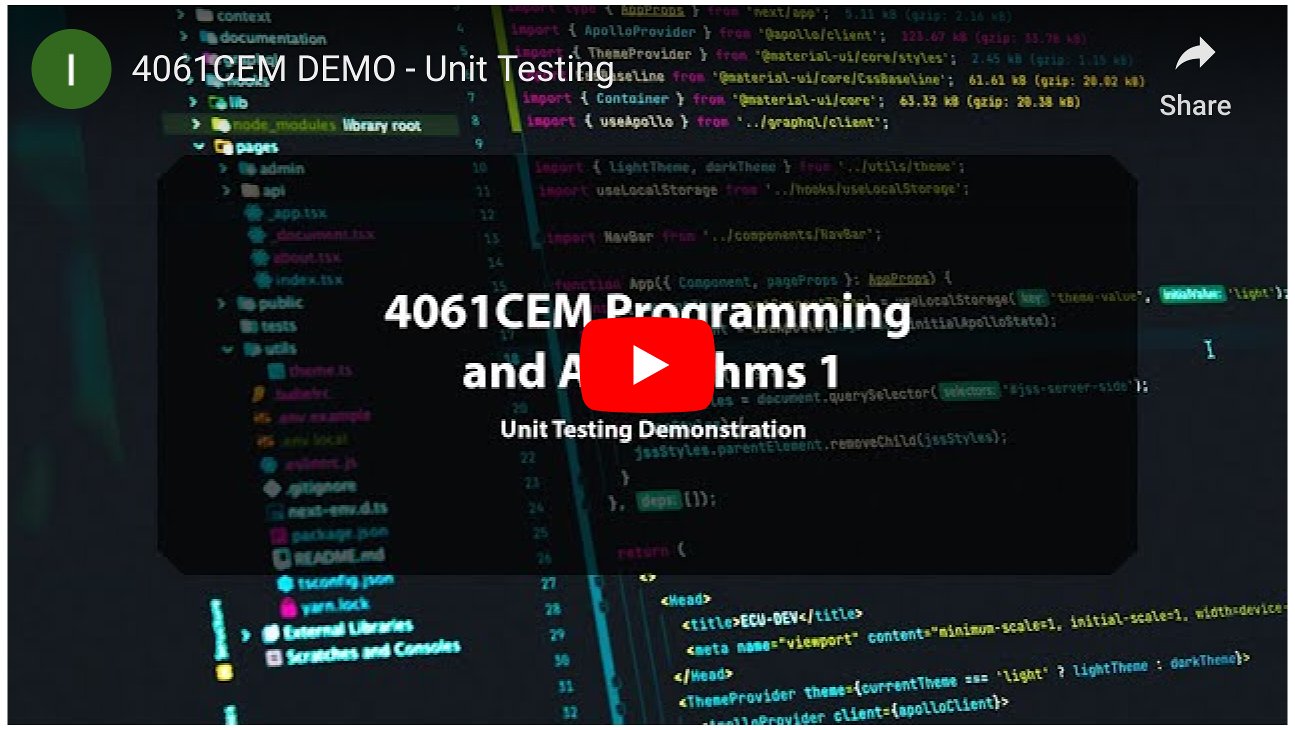
- Before you delve into writing your tests, consider the following questions:
    1. what do you want to test?
    2. are you writing a unit test or integration test?
- The structure of your test should loosely resemble:
    - create a set of inputs
    - execute the code that is being tested, and capture the output
    - compare the output with the expected result

# WRITING AN ASSERTION

- The last step to writing a test is validation of the output against the expected result, known as an **assertion**
- When it comes to writing an assertion, there are some best practices you should be following:
  - ensure the tests are repeatable
  - run the test multiple times to ensure you get the same output everytime
  - assert the results that relate to the input data

# WRITING AND EXECUTING UNIT TEST

- Demonstration of Unit Testing in Python
  - Refer to the pre-recorded video for a demonstration


I
4061CEM DEMO - Unit Testing
Share
4061CEM Programming and A hms 1
Unit Testing Demonstration

# WHAT ARE SIDE EFFECTS?

- Sometimes your code may not return a value from the function
- It may be the case that something will be altered outside the function
  - i.e. an attribute of a class, a file or a value in a database
- These are known as **side effects**, and should be considered before being included in the list of assertions
- If a unit of code has a lot of side effects, you are breaking the single responsibility principle

## SINGLE RESPONSIBILITY PRINCIPLE

- A programming principle that states the following:
  - Every module, class or function should have responsibility over a single part of a programs functionality
- For example, consider a function that compiles and prints a report:
  1. the content of the report could change
  2. the formatting of the report could also change
- These two aspects should be split into separate classes or functions
- Enables code to be designed in a way it is repeatable and simple for testing

# GOODBYE

- Questions?
  - Post them in the **Community Page** on Aula
- Contact Details:
  - Dr Ian Cornelius, ab6459@coventry.ac.uk