**Modular Design**

An important aspect of well-designed software is that programs are designed as a collection of modules. The term “module,” broadly speaking, refers to the design and/or implementation of specific functionality to be incorporated into a program. While an individual function may be considered a module, modules generally consists of a collection of functions (or other entities) [METIS, 4].

**Advantages of Modular Programming**

* Software Design: Provides a means for the development of well-designed programs
* Software Development: Provides a natural means of dividing up programming tasks. Provides a means for the reuse of the program code.
* Software Testing: facilitates the modification of specific program functionalities

**Why Test your Code?**

Testing proves that your code works as it is supposed to in situations it is designed to handle. It also good in situations where people use your program in unexpected ways. Writing tests gives the developers the confidence that their code works as more people begin to use your program. Developers can add new features to their programs and know that they have not broken existing behaviors.

**Unit Testing**

Unit testing is a level of software testing where individual units/ components of a software are tested. The purpose is to validate that each unit of the software performs as designed. A **unit** is the smallest testable part of any software. It usually has one or a few inputs and usually a single output. In procedural programming, a unit may be an ***individual program***, ***function***, ***procedure***, etc. In object-oriented programming, the smallest unit is a ***method***, which may belong to a base/ super ***class***, abstract class or derived/ child class. [**1**].

**Unit Testing** **Best Practices** [1]

* Find a tool/framework for your language.
* Focus on the test that impact the behavior of the system. DO NOT create test cases for everything.
* Isolate the development environment from the test environment.
* Use test data that is close to that of production.
* Before fixing a defect, write a test that exposes the defect. Why? First, you will later be able to catch the defect if you do not fix it properly. Second, your test suite is now more comprehensive. Third, you will most probably be too lazy to write the test after you have already fixed the defect.
* Write test cases that are independent of each other. For example, if a class depends on a database, do not write a case that interacts with the database to test the class. Instead, create an abstract interface around that database connection and implement that interface with a mock object.
* Aim at covering all paths through the unit. Pay particular attention to loop conditions.
* Make sure you are using a version control system to keep track of your test scripts.
* In addition to writing cases to verify the behavior, write cases to ensure the performance of the code.
* Perform unit tests continuously and frequently.

**Advantages and Disadvantages of Unit Testing [**3**]**

Advantages:

* Developers looking to learn what functionality is provided by a unit and how to use it can look at the unit tests to gain a basic understanding of the unit API.
* Unit testing allows the programmer to refactor code at a later date, and make sure the module still works correctly (i.e. Regression testing). The procedure is to write test cases for all functions and methods so that whenever a change causes a fault, it can be quickly identified and fixed.
* Due to the modular nature of the unit testing, we can test parts of the project without waiting for others to be completed.

Disadvantages:

* Unit testing can't be expected to catch every error in a program. It is not possible to evaluate all execution paths even in the most trivial programs.
* Unit testing by its very nature focuses on a unit of code. Hence it can't catch integration errors or broad system level errors.

**Test Runners in Python**

* **doctest**
* **unittest** [2]
* **nose** or **nose2** [4]
* **pytest** [5]

**Using doctest**

1. Run the module.py file shown below.

|  |
| --- |
| def square(x):  """Return the square of x.  >>> square(2)  4  >>> square(-2)  4  """  return x \* x  if \_\_name\_\_ == '\_\_main\_\_':  import doctest  doctest.testmod() |
| Filename: module.py |

1. Modify the content of the docttest part as shown below.

|  |
| --- |
| def square(x):  **"""**  **Return the square of x.**  **>>> square(2)**  **4**  **>>> square(-2)**  **5**  **"""**  return x \* x  if \_\_name\_\_ == '\_\_main\_\_':  import doctest  doctest.testmod() |
| Filename: module.py |

1. Run the module.py again. You should be seeing the below display.

|  |
| --- |
|  |
| **Running the module .py with test failure message** |

**unittest Python Unit Testing Framework**

**Note:** In this handout, version 3.7 is used. You can read the depth description of the API of unittest here: <https://docs.python.org/3.7/library/unittest.html>

The Python unit testing framework, sometimes referred to as “PyUnit,” is a Python language version of JUnit, by Kent Beck and Erich Gamma. JUnit is, in turn, a Java version of Kent’s Smalltalk testing framework. Each is the de facto standard unit testing framework for its respective language [**2**]

|  |  |
| --- | --- |
| **Table 1. Important Concepts** | |
| **Concept** | **Meaning** |
| **test fixture** | A test fixture represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process. |
| **test case** | A test case is the smallest unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class, TestCase, which may be used to create new test cases. |
| **test suite** | A test suite is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together. |
| **test runner** | A test runner is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests. |

|  |  |
| --- | --- |
| **Table 2. Assert methods in unittest.TestCase class** | |
| **Method** | **Equivalent to** |
| **.assertEqual(a, b)** | **a == b** |
| **.assertTrue(x)** | **bool(x) is True** |
| **.assertFalse(x)** | **bool(x) is False** |
| **.assertIs(a, b)\*** | **a is b** |
| **.assertIsNone(x)\*** | **x is None** |
| **.assertIn(a, b)\*** | **a in b** |
| **.assertIsInstance(a, b)\*** | **isinstance(a, b)** |

\* This functions have the equivalent .assertIsNot, assertIsNotNone, etc.

|  |  |
| --- | --- |
| **Table 3. Ways to Execute the unittest Test Runner** | |
| **Method** | **Description** |
| 1. python testcode.py | Good for single test code. |
| 1. python -m unittest testcode | Takes a module in the standard library or on sys.path and runs it as a script.  Here, the module name is testcode which is referring to the testcode.py |
| 1. python -m unittest testcode -v | Display verbose test result(s), i.e. -v will display the name of the test module(s) |
| 1. python -m unittest -h | Lists all the command-line options |
| 1. python -m unittest discover | Allows auto-discovery of the module (i.e. the python file) containing tests. This option requires that the file should start with the word ‘test’ (e.g. testcode.py, test\_wedding.py, etc.)  You can add the -v option to display the name of the test modules. |
| 1. python -m unittest discover -s testcode | Good for running multiple files in a directory. Here testcode is the directory.  You can add the -v option to see the module names. |
| 1. python -m unittest discover -s tests -t src | Allows the unittest to look for the unittest source codes in the src directory. |

**Test Cases**

Test cases are considered the basic building blocks of unit testing. A test case is a single scenario that must be set up and checked for correctness.

In unittest Python Framework, test cases are instances of the TestCase class.

**Mocking**

Mocking is the process of declaring within a test that a certain function call should be stipulated to give a particular output, and the function call itself should be suppressed. unittest.mock (Python 3.3 above)

A Mock is a fake, stub or a replacement of some existing object, variable or callable that return a pre-determined response.

|  |
| --- |
| Mock Testing Usage Scenario |
| Imagine that we have a unit test that is responsible for testing the user registration feature of our application. Obviously, we want our test to make sure that a user can be registered, but do we really want the test to create a real user in our database each time we run it?  If we run our tests thousands of times, it would mean that our test would create a new user in our database for each and every time. That would not only be annoying to deal with from a data perspective, but it would also slow down our tests since it will be writing to the database all the time. [ |

**Hands-on Exercises**

1. **Python assert Statement**

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program. Almost every programming language includes an assert statement.

The Python assert statement can be used in two ways:

* assert statement has a condition and if the condition is not satisfied the program will stop and give AssertionError.
* assert statement can also have a condition and an optional error message. If the condition is not satisfied assert stops the program and gives AssertionError along with the error message.

1. **Using py.test**

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program

1. You need to install pytest since it is not the standard python library for unit testing. See below command

pip install pytest

|  |  |
| --- | --- |
|  | A screenshot of a cell phone  Description automatically generated |
| 1. Display after running the **help(unittest)** command | 1. Simple Usage of the unittest Test Runner |

1. **Using unittest**
2. Run python in the terminal and execute the following lines:

>> import unittest

>> help(unittest)

You will see the below display. Take time to read the ‘***Help on package unittest***’

|  |  |
| --- | --- |
|  | A screenshot of a cell phone  Description automatically generated |
| 1. Display after running the **help(unittest)** command | 1. Simple Usage of the unittest Test Runner |

1. Download the python codes here: <https://bit.ly/2yc2ohL>
2. Run your python terminal or anaconda command prompt and activate your anaconda softDesLec environment. Navigate to your unit test source codes as shown in the below screengrab. Run Visual Studio Code and open the **unit\_testing** folder as shown below:

|  |  |
| --- | --- |
| A screenshot of a social media post  Description automatically generated | A screenshot of a computer screen  Description automatically generated |
| 1. Preliminary unittest test codes | 1. Opening the entire unit\_testing folder in Visual Studio Code |

1. Run the testcode.py in the terminal. You will see the same output as shown below.

A screenshot of a social media post

Description automatically generated

Try executing the test runner using the options b, c, d, and e in Table 3. Take time to observe the display.

1. Run the following command in your terminal.

* cp testcode.py testcode\_01.py

1. Execute the test runner using the Table 3 option (f). Add the -v option. You should have the same display in the below screengrab (a).
2. Run the below Linux commands. Execute Table 3 option (g). You should have the same display in the below screengrab (b).

* mkdir src
* mv test\* src

|  |  |
| --- | --- |
|  |  |
| 1. Running the test runner using Table 3 option (f) | 1. Running the test runner using Table 3 option (g) |

1. **Adding test methods**
2. Create a new part2 folder and copy the files: names.py, full\_names.py, and test\_full\_names.py. Open the part2 folder in Visual Studio code.
3. Run the full\_names.py using the unittest test runner options previously mentioned. You should have the same display as shown below.

Note: The unittest output also includes the test function and the beginning of the docstring.

|  |  |
| --- | --- |
|  |  |
| 1. Running the test\_full\_names module using -m and -v | 1. Running the test\_full\_names module using -m, -v and discover |

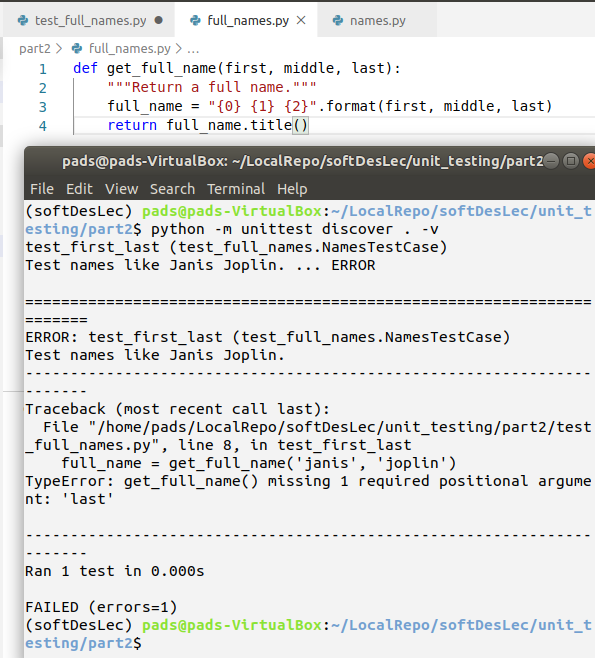
**Test Failures and Errors**

You can observe what happens when a test fails by simply changing the test's condition so that it will intentionally fail. Failing tests are important; they tell you that a change in the code has affected existing behavior. In general, you should not modify a test once it’s written. When a test fails, you need to modify the test code so the existing behavior still works. A test that raises AssertionError is considered to have failed, whereas a test that raises any exception other than AssertionError is considered to be in error.

1. Modify the get\_full\_name method of the full\_names.py as show in (b).

|  |  |
| --- | --- |
| def get\_full\_name(first, last):  """Return a full name."""  full\_name = "{0} {1}".format(first, last)  return full\_name.title() | def get\_full\_name(first, middle, last): """Return a full name.""" full\_name = "{0} {1} {2}".format(first, middle, last) return full\_name.title() |
| 1. Current get\_full\_name method | 1. Modified get\_full\_name method |

Run again the test runner code. This will produce the below error.



1. Modify the get\_full\_name method as shown below.

def get\_full\_name(first, last, middle=''):

"""Return a full name."""

if middle:

full\_name = "{0} {1} {2}".format(first, middle, last)

else:

full\_name = "{0} {1}".format(first,last)

return full\_name.title()

**Skipped Tests**

The unittest module provides two decorators: skipIf and skipUnless. The unittest.Testloader class is used this time.

**Loading Tests**

This is for large application that may contain several unit test modules.

**Adding more Tests**

1. Add the new test method shown below. Run the test and observe the output.

|  |
| --- |
| import unittest  from full\_names import get\_full\_name  class NamesTestCase(unittest.TestCase):  """Tests for names.py."""    def test\_first\_last(self):  """Test names like Janis Joplin."""  full\_name = get\_full\_name('janis', 'joplin')  self.assertEqual(full\_name, 'Janis Joplin')  def test\_middle(self):  """Test names like David Lee Roth."""  full\_name = get\_full\_name('david','roth', 'lee')  self.assertEqual(full\_name,'David Lee Roth')  if \_\_name\_\_=='\_\_main\_\_':  unittest.main() |

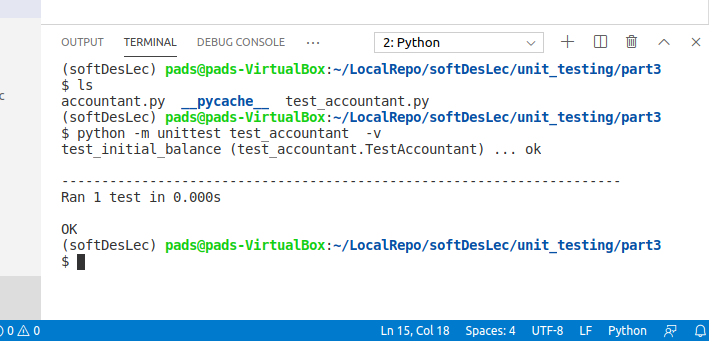
1. **Testing A Class**

Testing a class is similar to testing a function, since you will mostly be testing your methods.

**setUp() and tearDown() Functions**

These functions allow you to define instructions that will be executed before and after each test method

1. Open the part3 folder in Visual Studio code. Run the test\_accountant module as shown below.



1. **Using unittest.mock**

* Decorator
* Context Manager
* Inline

**To Do Exercise**

* Modify the code.py so that it will return the double of a given number. You have to add a new test method in your testcode.py and you should be able to capture at below four test cases in your method. Name the method **testdouble**. You will have the same display as shown in (a). Now, modify the last case so that your display will have the same error in (b).

Test Cases:

1. 2 \* 2 = 4
2. 4 \* 2 = 8
3. 0 \* 2 = 0
4. -2 \* 2 = -4

|  |  |
| --- | --- |
|  |  |
| 1. Running the testcode.py with no error | 1. Running the testcode.py with error |

**References**:

Websites:

1. <http://softwaretestingfundamentals.com/unit-testing/>
2. <https://docs.python.org/2/library/unittest.html>
3. <https://www.guru99.com/unit-testing-guide.html>
4. <https://nose.readthedocs.io/en/latest/testing.html>
5. <https://docs.pytest.org/en/latest/>
6. <https://realpython.com/python-testing/>
7. https://coderbook.com/@marcus/write-unit-tests-with-pythons-unittest-module/

METIS Books:

1. **Core Python Programming**, ISBN: 9789351198918,

assert Python Statement and Assertion Error (page 140)

1. **Professional Python**, ISBN: 9781119070832,

assert Python Statement and Assertion Error (Chapter 11)

1. **Fundamentals of Python: First Programs**. ISBN: 9781337671019

Testing in the Implementation and Integration Phases (page 36)

1. **Introduction to Computer Science Using Python: A Computational Problem-Solving Focus**. ISBN: 9780470912041

Unit Testing - pages 247, 269, 274, 276, page 289 (unit testing using test drivers), and pages 318  to 324 (unit testing on text files)

1. **Professional Python** ISBN: 9781119070832

Unit Testing, (Chap 11)