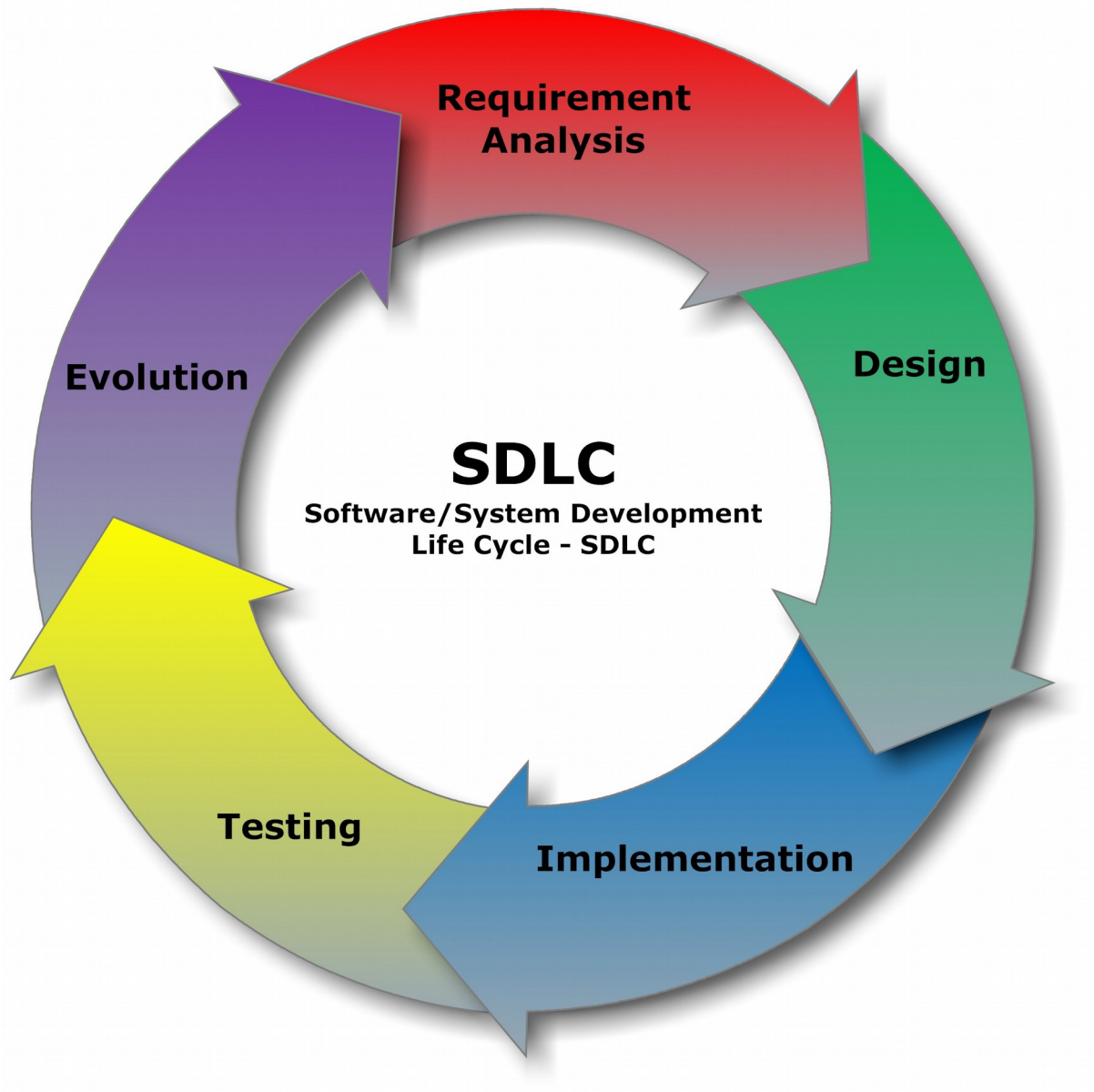
Introduction to Unit Testing in Python

Eric Solis Montufar

The Software Development Life Cycle

Unit testing is part of the Software Development Life Cycle (SDLC). The SDLG is a process used in software development to design, develop and test software, and its purpose is to produce high-quality software that fulfills the expectations and adjusts to the times and cost estimates.



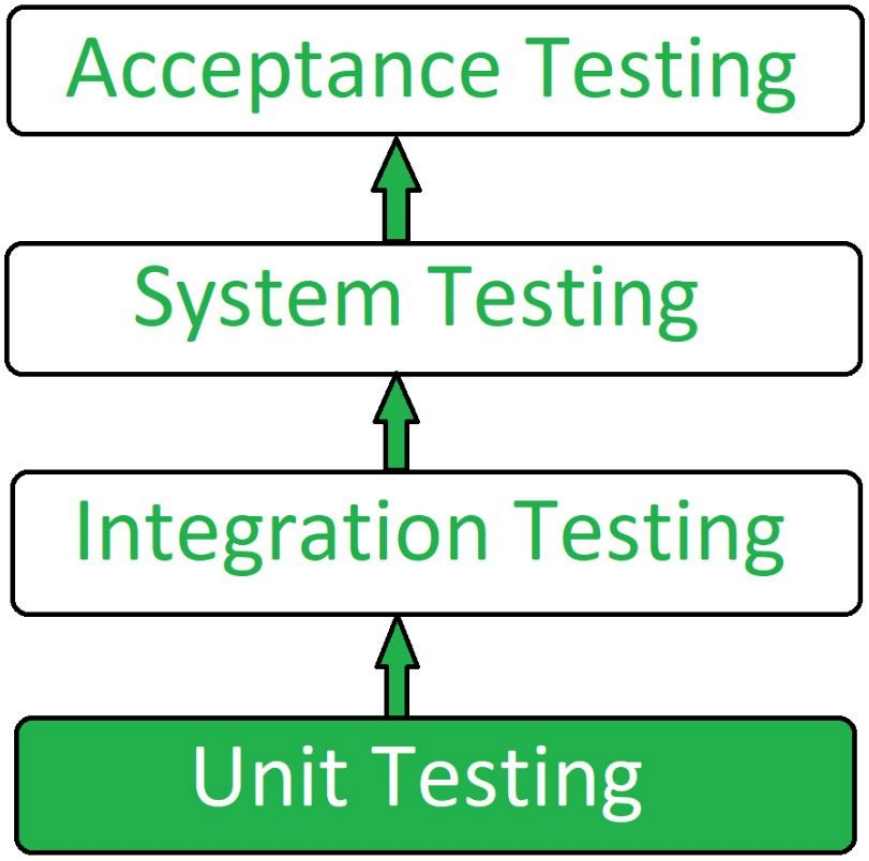
Testing Phase in the SDLC

The main goal of the testing stage is to report, monitor and retest software until they reach the quality standards defined in the requirements. Testing helps to catch errors in the code at the early stages to minimize the cost and time of resolving them.

The testing phase is an important element of the SDLC. These processes are executed sequentially and methodically to ensure the fulfillment of all the requirements of the software application. This stage is usually divided into four main phases:

- Unit Testing: this is a test performed in smaller components of the code. This is usually done to single functions or methods to ensure that each unit of software is working as it should.
- Integration Testing: This stage of the testing phase focuses on testing different modules working together. This step is necessary to make sure that the integration of the modules are working properly.
- System Testing: System testing is the stage where the parts are put together to test them as a single piece of software. Here, the interaction of the pieces is evaluated as well as the performance, reliability, and security of the integrated system.
- User acceptance Test: Here, customers evaluate the components of the application to see whether it meets their requirements. At this point, users can ask the development team to do modifications if needed.

In this article, we will only focus on the testing stage of the SDLC.



Working on an application without unit test, may turn into a very difficult task to make a major change or addition to your project later on down the road without introducing other problems. With the unit tests in place, you can confidently add functionality to your project later on and ensure that all the previous components are still valid.

Hands on

First, we have to create the code we want to test. For this, we are going to use a simple calculator with the basic operations (addition, subtraction, multiplication and division).

```python
class Calculator:
    def add(x, y):
    #     Addition Function
        return x + y


    def subtract(x, y):
    #     Subtraction Function
        return x - y


    def multiply(x, y):
    #     Multiplication Function
        return x * y


    def divide(x, y):
    #     Division Function
        if y == 0:
            raise ValueError('Can not divide by zero!')
        return x / y
```

Once we have the python file we want to test. We can start to define the cases we want to test and work on the testing script.
For this:

1. Import the unittest library and the class from the python file we want to test.
2. Define a class with the tester itself extending from unittest.TestCase.
3. Define the methods for testing every module of your code. For this example, we test the four operations included in our calculator app.
4. Make sure to test with different cases that could break your code.

The TestCase class provides several assert methods to check for and report failures. The following table lists the most commonly used 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

There are many types of assertions supported in Python 3.7. See https://docs.python.org/3/library/unittest.html#assert-methods.

Here, we are going to use the assertEqual() statement to test whether the result of the modules is equal to the expected result for each operation. In a separate python file we create a python script for the tester.

```python
import unittest
from Calculator import Calculator as calc


class TestCalc(unittest.TestCase):


    def test_add(self):
        self.assertEqual(calc.add(1, 5), 6)
        self.assertEqual(calc.add(-1, 2), 1)
        self.assertEqual(calc.add(-1, -3), -4)

    def test_subtract(self):
        self.assertEqual(calc.subtract(10, 2), 8)
        self.assertEqual(calc.subtract(-1, 3), -4)
        self.assertEqual(calc.subtract(-1, -1), 0)

    def test_multiply(self):
        self.assertEqual(calc.multiply(10, 3), 30)
        self.assertEqual(calc.multiply(-1, 2), -2)
        self.assertEqual(calc.multiply(-1, -2), 2)

    def test_divide(self):
        self.assertEqual(calc.divide(10, 5), 2)
        self.assertEqual(calc.divide(-1, 1), -1)
        self.assertEqual(calc.divide(-1, -1), 1)
        self.assertEqual(calc.divide(5, 2), 2.5)


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

Once we run out tester, it will perform the unit tests that we defined for each of our functions in the Calculator class.

When one of the units test fail, we see something like this:

```
======================================================================
FAIL: test_add (__main__.TestCalc)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "tester.py", line 10, in test_add
    self.assertEqual(calc.add(-1, 2), -1)
AssertionError: 1 != -1

----------------------------------------------------------------------
Ran 4 tests in 0.000s
```

When the tests succeeded, the see just a summary of the performed tests like this:

```
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

Conclusions

The unit test is an essential stage in the software development life cycle. This plays an important role when adding functionalities or modifying existing pieces of code. As we saw in this introduction to unit test, implementing a tester can be done easily with the unittest library in python.