Unit Testing

Unit Testing is the first level of software testing where the smallest testable parts of a software are tested. This is used to validate that each unit of the software performs as designed. Suppose, you need to test your project. You know what kind of data the function will return. After writing huge code, you need to check it whether the output is correct or not.

Normally, what we do is printing the output and match it with the reference output file or check the output manually.

To reduce this pain, Python has introduced unittest module. Using this module you can check the output of the function by some simple code. A unit test consists of one or more assertions.The unittest.TestCase class contains a number of assert methods, so be sure to check the list and pick the appropriate methods for your tests.

Assertion

- Assertions are simply boolean expressions that checks if the conditions return true or not. If it is true, the program does nothing and move to the next line of code. However, if it's false, the program stops and throws an error.
- It is also a debugging tool as it brings the program on halt as soon as any error is occurred and shows on which point of the program error has occurred.

OOP concepts supported by unittest framework:

- **test fixture:**
  A test fixture is used as a baseline for running tests to ensure that there is a fixed environment in which tests are run so that results are repeatable.
- **test case:**
  A test case is a set of conditions which is used to determine whether a system under test works correctly.
- **test suite:**
  Test suite is a collection of testcases that are used to test a software program to show that it has some specified set of behaviours by executing the aggregated tests together.
- **test runner:**
  A test runner is a component which set up the execution of tests and provides the outcome to the user.

Types of possible test outcomes

This unittest has 3 possible outcomes. They are mentioned below:

1. **OK:** If all test cases are passed, the output shows OK.
2. **Failure:** If any of test cases failed and raised an AssertionError exception
3. **Error:** If any exception other than AssertionError exception is raised.

There are several function under unittest module. They are listed below.

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
assertIs(a,b)	a is b
assertIs(a,b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

Advantages of using Python Unit testing

- It helps you to detect bugs early in the development cycle
- It helps you to write better programs
- It syncs easily with other testing methods and tools
- It will have many fewer bugs
- It is easier to modify in future with very less consequence

Example 1

```
import unittest
class TestStringMethods(unittest.TestCase):
        def test_upper(self):
                self.assertEqual('foo'.upper(), 'FOO')
        def test_isupper(self):
                self.assertTrue('FOO'.isupper())
                self.assertFalse('Foo'.isupper())
        def test_split(self):
                s = 'hello world'
                self.assertEqual(s.split(), ['hello', 'world'])

if __name__ == '__main__':#It checks if a module is being imported or not.__name__ (inbuilt attribute) it is the module's filename
        unittest.main()
```

- A testcase is created by subclassing unittest.TestCase. The three individual tests are defined with methods whose names start with the letters test. This naming convention informs the test runner about which methods represent tests.
- The paricular point of each test is a call to assertEqual() to check for an expected result; assertTrue() or assertFalse() to verify a conditionThese methods are used instead of the assert statement so the test runner can accumulate all test results and produce a report.
- The final block shows a simple way to run the tests. unittest.main() provides a command-line interface to the test script.

- When run from the command line, the above script produces an output that looks like this:

Output

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Passing the -v option to your test script will instruct unittest.main() to enable a higher level of verbosity, and produce the following output:

Output

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Example 2

```
import unittest
from math import factorial

class TestFactorial(unittest.TestCase):
    """
    Our basic test class
    """

    def test_fact(self):
        """
        The actual test.
        Any method which starts with ``test_`` will considered as a test case.
        """
        res = factorial(5)
        self.assertEqual(res, 120)

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

Debugging

Debugging is an important step of any software development project. The Python debugger `pdb` implements an interactive debugging environment that you can use with any of your programs written in Python.

With features that let you pause your program, look at what values your variables are set to, and go through program execution in a discrete step-by-step manner, you can more fully understand what your program is doing and find bugs that exist in the logic or troubleshoot known issues.

Table of Common pdb Commands

Here is a table of useful `pdb` commands along with their short forms to keep in mind while working with the Python debugger.

Command	Short form	What it does
`args`	`a`	Print the argument list of the current function
`continue`	`c` or `cont`	Continues program execution
`help`	`h`	Provides list of commands or help for a specified command
`list`	`l`	Print the source code around the current line
`next`	`n`	Continue execution until the next line in the current function is reached or returns
`step`	`s`	Execute the current line, stopping at first possible occasion
`quit` or `exit`	`q`	Aborts the program
`return`	`r`	Continue execution until the current function returns

Example 1

```
#experiment with the Python debugger, pdb
import pdb#import python debugger module
a = "aaa"
pdb.set_trace()#To trace
b = "bbb"
c = "ccc"
d="34"
final = a + b + c + d
print final
```

1. Now run your program from the command line as you usually do, which will probably look

   PROMPT> python debugging.py
2. When your program encounters the line with pdb.set_trace() it will start tracing. That is, it will (1) stop, (2) display the “current statement” (that is, the line that will execute next) and (3) wait for your input. You will see the pdb prompt, which looks like this
   (Pdb)
3. **next(n) command**-At the (Pdb) prompt, press the lower-case letter “n” (for “next”) on your keyboard, and then press the ENTER key. This will tell pdb to execute the current statement. Keep doing this — pressing “n”, then ENTER. Eventually you will come to the end of your program, and it will terminate and return you to the normal command prompt.

4. This time, do the same thing as you did before. Start your program running. At the (Pdb) prompt, press the lower-case letter “n” (for “next”) on your keyboard, and then press the ENTER key again and again.This will repeat last debugging command only that is next(n) command
5. **quit(q) command**-When you see the (Pdb) prompt, just press “q” (for “quit”) and the

ENTER key. Pdb will quit and you will be back at your command prompt.

6. **print(p)command-**When you see the (Pdb) prompt, enter “p” (for “print”) followed by the name of the variable you want to print and press enter buttin.You can print multiple variables, by separating their names with commas.
   Example: p a, b, c
7. **Continue(c)-**You probably noticed that the “q” command got you out of pdb in a very crude way — basically, by crashing the program.If you wish simply to stop debugging, but to let the program continue running, then you want to use the “c” (for “continue”) command at the (Pdb) prompt
8. **list(l) command-**“l” shows you, on the screen, the general area of your program’s souce code that you are executing. By default, it lists 11 (eleven) lines of code. The line of code that you are about to execute (the “current line”) is right in the middle, and there is a little arrow “–>” that points to it.

Example 2

```
import pdb
def combine(s1,s2):
    s3 = s1 + s2 + s1
    return s3
a = "aaa"
pdb.set_trace()
b = "bbb"
c = "ccc"
final = combine(a,b)
print final
```

step into(s) command-This command is used to step into the function. If you press ‘s’ after the (final=combine(a,b))function call then the next statement that pdb would show you would be the first statement in the combine function.And you will continue debugging from there.

Return(r) command-If you have to step through a lot of uninteresting code in the function.If you want just to continue to the end of the function, then resume your stepping through the code.The command to do it is “r”.