**Automated Testing**

Even though you explored and learned how to debug applications when errors are reported, you would prefer not having to find errors in our applications. To increase the chances of having a bug-free code base, most developers rely on automated testing.

At the beginning of their careers, most developers will just manually test their code as they develop it. By just providing a set of inputs and verifying the output of the program, you can get a basic level of confidence that our code "works." But this quickly becomes tedious and does not scale as the code base grows and evolves. Automated testing allows you to record a series of steps and stimuli that you perform in our code and have a series of expected output recorded.

This is extremely efficient to reduce the number of bugs in our code base, because not only are we verifying the code, but we are also implementing it, and you keep a record of all those verifications for future modifications of the codebase.

The amount of test lines that you write for each line of code really depends on each application.

**Test Categorization**

One of the first things to think about when writing an automated test is "What are we verifying?". And that would depend on the "level" of testing that you are doing. There is a lot of literature about how to categorize different test scenarios in the functions that they validate and the corresponding dependencies they have. It is not the same to write a test that just validates a simple Python function in our source code, as it is to write something that validates an accounting system that connects to the internet and sends emails. To validate large systems, it is common to create different types of tests. They are usually known as the following:

Unit tests: These are tests that just validate a small part of your code. Usually, they just validate a function with specific inputs within one of your files and only depend on code that has already been validated with other unit tests. System Integration tests (SIT): These are more coarse-grained tests that will either validate interactions between different components of your codebase (known as integration tests without environment) or the interactions between your code and other systems and the environment (known as integration tests with the environment). Functional or end-to-end tests / User Acceptance Testing (UAT): These are usually really high-level tests that depend on the environment and often on external systems that validate the solution with inputs as the user provides them.

Software developers tend to prefer unit tests as they don't have external dependencies and are more stable and faster to run

**Test Coverage**

Something that generates debate across the community is test coverage. When you write tests for our code, you start to exercise it and begin to hit different code paths. As you write more tests, we cover more and more of the code that you are testing. The percentage of code that you test is known as test coverage, and developers will argue that different percentages are "the right amount." Getting to 100% coverage might seem an unnecessary task, but it proves to be quite useful in large codebases that need to perform tasks such as migrating from Python 2 to Python 3. However, this all depends on how much you are willing to invest in testing your application, and each developer might target a different number for each of the projects that they run.

Moreover, something important to remember is that 100% coverage does not mean that your code does not have bugs. You can write tests that exercise your code but do not properly validate it, so be mindful of falling into the trap of just writing tests to hit the coverage target. Tests should be written to exercise the code with inputs that will be provided by users and try to find edge cases that can uncover issues with the assumptions that you made at the time that you wrote it, and not just to hit a number.

**Writing Tests in Python with Unit Testing**

The Python standard library comes with a module, unittest, to write test scenarios and validate your code. Usually, when you are creating tests, we create a file for the test to validate the source code of another file. In that file, you can create a class that inherits from unittest.TestCase and has method names that contain the word test to be run on execution. You can record expectations through functions such as assertEquals and assertTrue, which are part of the base class, and you can, therefore, access them.

**Unittest**

unittest is the testing module available in the Python standard library. Its API calls are similar to JUnit/nUnit/CppUnit series of tools, thus makes it familiar to use. It's documentation is available at <http://docs.python.org/3/library/unittest.html>. One of the most import aspects of unittest is to evaluate certain test conditions, based on which tests can bet set to pass or fail. In python various assert helps us in achieving it.

**What is assert ?**

assert\*'s can be treated as condition validators which allows automation engineers an easy way to validate certain test conditions and pass/fail test cases in a single line of code. Basic example of it is shown below.

**%%**writefile num.py

*#meant only for your jupyter notebook*

​

**def** multiply(a,b):

**return** a**\***b

​

​

​

​

​

Overwriting num.py

**%%**writefile test\_num.py

​

**import** unittest *#module to write testcases*

**from** num **import** multiply

​

​

​

**class** TestNum(unittest.TestCase):

**def** test\_numbers\_3\_4(self):

result **=** multiply(3,4) *#test the function*

self.assertEqual(result,12)

**def** test\_characters\_a\_3(self):

result **=** multiply('a',3)

self.assertEqual(result,'aaa')

**def** test\_characters(self):

result **=** multiply('a','b')

self.assertEqual(result,'ab')

​

**if** \_\_name\_\_ **==** '\_\_main\_\_':

unittest.main()

Overwriting test\_num.py

*#it can be executed even in the command prompt*

​

*#To run the code in the jupyter itself by prefixing ! symbol and if we want more information on the methods and functions*

*#we can provide -v option*

**!** python test\_num.py **-**v

​

​

test\_characters (\_\_main\_\_.TestNum) ... ERROR

test\_characters\_a\_3 (\_\_main\_\_.TestNum) ... ok

test\_numbers\_3\_4 (\_\_main\_\_.TestNum) ... ok

======================================================================

ERROR: test\_characters (\_\_main\_\_.TestNum)

----------------------------------------------------------------------

Traceback (most recent call last):

File "test\_num.py", line 19, in test\_characters

result = multiply('a','b')

File "C:\Users\asha.t\AppData\Local\Programs\Python\Python37-32\Python - Training\15 - Unit testing\num.py", line 5, in multiply

return a\*b

TypeError: can't multiply sequence by non-int of type 'str'

----------------------------------------------------------------------

Ran 3 tests in 0.016s

FAILED (errors=1)

​

**%%**writefile cap.py

**def** cap\_text(text):

**return** text.capitalize()

Overwriting cap.py

**%%**writefile test\_cap.py

**import** unittest

**import** cap

​

**class** TestCap(unittest.TestCase):

**def** test\_one\_word(self):

text **=** 'python'

result **=** cap.cap\_text(text)

self.assertEqual(result, 'Python')

**def** test\_multiple\_words(self):

text **=** 'monty python'

result **=** cap.cap\_text(text)

self.assertEqual(result, 'Monty Python')

**if** \_\_name\_\_ **==** '\_\_main\_\_':

unittest.main()

Overwriting test\_cap.py

**!** python test\_cap.py **-**v

test\_multiple\_words (\_\_main\_\_.TestCap) ... ok

test\_one\_word (\_\_main\_\_.TestCap) ... ok

----------------------------------------------------------------------

Ran 2 tests in 0.000s

OK

What happened? It turns out that the .capitalize() method only capitalizes the first letter of the first word in a string. Doing a little research on string methods, we find that .title() might give us what we want.

**%%**writefile cap.py

**def** cap\_text(text):

**return** text.title() *# replace .capitalize() with .title()*

​

*#after making changes in the code , go and execute the testing cases and we can find the results are ok*

Overwriting cap.py

Create a function, is\_divisible, which checks whether a number is divisible by another. Save this function in a file named sample\_code. This function is also provided in the sample\_code.py file. The file just has a single function that checks whether a number is divisible by another:

**%%**writefile sample\_code.py

**def** is\_divisible(x, y):

**if** x **%** y **==** 0:

**return** **True**

**else**:

**return** **False**

Overwriting sample\_code.py

​

​

Create a test file that will include the test cases for our function. Then, add the skeleton for a test case:

We will now define a test, the test should be pre-fixed with test\_ for unittest to find them. In fact unittest uses several conventions to find tests, which are: • Search for test\_\*.py or \*\_test.py files. • From those files, collect test items: – test\_prefixed test functions, – test\_prefixed test methods inside Test prefixed test classes (without an\_\_init\_\_method)

**%%**writefile test\_division.py

**import** unittest

**from** sample\_code **import** is\_divisible

​

**class** TestIsDivisible(unittest.TestCase):

**def** test\_divisible(self):

self.assertTrue(is\_divisible(10,2))

self.assertTrue(is\_divisible(10, 10))

self.assertTrue(is\_divisible(1000, 1))

**def** test\_not\_divisible(self):

self.assertFalse(is\_divisible(15, 3))

self.assertFalse(is\_divisible(5, 6))

self.assertFalse(is\_divisible(10, 3))

unittest.main()

Overwriting test\_division.py

**!** python test\_division.py **-**v

test\_divisible (\_\_main\_\_.TestIsDivisible) ... ok

test\_not\_divisible (\_\_main\_\_.TestIsDivisible) ... FAIL

======================================================================

FAIL: test\_not\_divisible (\_\_main\_\_.TestIsDivisible)

----------------------------------------------------------------------

Traceback (most recent call last):

File "test\_division.py", line 13, in test\_not\_divisible

self.assertFalse(is\_divisible(15, 3))

AssertionError: True is not false

----------------------------------------------------------------------

Ran 2 tests in 0.006s

FAILED (failures=1)

1.Method               -- read accordingly

2.Checks that

3.meaning

​

1.assertEqual(a=3, b=2)

2.a == b

3.Test that first and second evaluate to same value

​

​

assertNotEqual(a=3, b=2)

a != b

Test that first and second evaluate to different value

​

​

assertTrue(x)

bool(x) is True

Test that first evaluates to True

​

​

assertFalse(x)

bool(x) is False

Test that first evaluates to False

​

​

assertIs(a, b)

a is b

Test that first and second evaluate to the same object.

​

​

assertIsNot(a, b)

a is not b

Test that first and second or don’t evaluate to the same object.

​

​

assertIsNotNone(x)

x is not None

Test that first do not evaluates to None

​

​

assertIsNone(x)

x is None

Test that first evaluates to None

​

​

assertIn(a, b)

a in b

Test that first evaluates to be present in second collection

​

​

assertNotIn(a, b)

a not in b

Test that first evaluates to be not present in second collection

​

​

assertIsInstance(a, b)

isinstance(a, b)

Test that first evaluates to be instance of second

​

​

assertNotIsInstance(a, b)

not isinstance(a, b)

Test that first evaluates to be not instance of second

​

​

assertAlmostEqual(a, b)

a is almost equal b

Test that first is nearly equal to second

​

​

assertNotAlmostEqual(a, b)

a is not almost equal b

Test that first is not nearly equal to second

​

​

assertGreater(a, b)

a > b

Test that first is greater than second

​

​

assertGreaterEqual(a, b)

a >= b

Test that first is greater or equal to second

​

​

assertLess(a, b)

a < b

Test that first is less than second

​

​

assertLessEqual(a, b)

a <= b

Test that first is less or equal to second

**import** unittest

a **=** d **=** 10

aa **=** 1111

bb **=** 1111

b **=** [12]

c **=** [10]

​

**from** unittest **import** TestCase

tc **=** unittest.TestCase('\_\_init\_\_') *#create an object of the unittest Testcase class by calling its init method*

assertEqual assertEqual validates weather value of first and second variables is same. It DO NOT check if items are same, but just the value.

tc.assertEqual(aa,bb)

​

*# assertEqual*

*# assertEquat validates weather value of first and second variables is same. It DO NOT check if items are same, but just the value.*

**try**:

tc.assertEqual(a,b)

**except** AssertionError **as** e:

print('Assertionerror:',e) *#the error number*

Assertionerror: 10 != [12]

a **=** d **=** 10

b **=** [12]

c **=** [10]

**try**:

print(tc.assertEqual(a, c))

**except** AssertionError **as** a:

print("AssertionError: ", a)

AssertionError: 10 != [10]

a **=** d **=** 10

b **=** [12]

c **=** [10]

tc.assertEqual(a, d)

assertNotEqual assertNotEquat validates whether value of first and second variables are not same. Again it also DO NOT check if items are not same, but just the value are different. Although common sense dictates us that if the items have different values they are can not be same :)

tc.assertNotEqual(a, aa) *#no error so no output*

**try**: *#if no try except then the will be stack trace error message displayed*

tc.assertNotEqual(a,d)

**except** AssertionError **as** e:

print(e)

10 == 10

assertTrue

assertTrue validates whether the value of item **is** **True**.

tc.assertTrue(a **==** a)

*# When using conditions, make sure that they are valid, otherwise asserts will fail, as shown in the below example.*

**try**:

tc.assertTrue(a **<** b)

**except** AssertionError **as** e:

print(e)

**except** TypeError **as** e:

print(e)

'<' not supported between instances of 'int' and 'list'

​

**try**:

print(tc.assertTrue(a **==** c))

**except** AssertionError **as** e:

print(e)

**try**:

print(tc.assertTrue(a, d))

**except** Exception **as** e:

print(e)

assertFalse

assertTrue validates whether the value of item **is** **False**.

**try**:

tc.assertFalse(a **==** a)

**except** AssertionError **as** e:

print(e)

True is not false

When using conditions, make sure that they are valid, otherwise asserts will fail, **as** shown **in** the below example.

**try**:

tc.assertFalse(a **>** b)

**except** TypeError **as** e:

print(e)

'>' not supported between instances of 'int' and 'list'

When the validating condition results **in** **False** than, assertFalse do **not** **raise** any exception.

**try**:

tc.assertFalse(a **==** c)

tc.assertFalse(a **<** d)

**except** AssertionError **as** e:

print(e)

assertIs

It asserts **if** first **and** second element are same.

print(tc.assertIs(a, a)) *#both as same so it shows there is no difference*

None

**try**:

print(tc.assertIs(a,b))

**except** AssertionError **as** e:

print(e)

10 is not [12]

**try**:

print(tc.assertIs(a,c))

**except** AssertionError **as** e:

print(e)

**try**:

print(tc.assertIs(a,d)) *#a and d are same*

**except** AssertionError **as** e:

print(e)

None

assertIsNot

It asserts **if** first **and** second element are **not** same.

**try**:

print(tc.assertIsNot(a, a))

**except** AssertionError **as** e:

print(e)

unexpectedly identical: 10

print(tc.assertIsNot(a,b)) *#both are different*

None

print(tc.assertIsNot(a,c))

**try**:

print(tc.assertIsNot(a,d))

**except** Exception **as** e:

print(e)

assertIsNone

assertIsNone asserts that the value of element **is** **None**

**try**:

tc.assertIsNone(a**/**a)

**except** Exception **as** e:

print(e)

1.0 is not None

assertIsNotNone

assertIsNone asserts that the value of element **is** notNone

tc.assertIsNotNone(a) *#does not do any thing if it is none*

assertIn

assertIn asserts that element **is** present **in** the collection.

tc.assertIn(a, [a])

*# Checking list*

x **=** [1, 2, 3, 4]

tc.assertIn(1, x)

*# Asserting dictionary keys*

x **=** {1: 2, 2: 3}

**try**:

tc.assertIn(1, x)

**except** AssertionError **as** e:

print(e)

assertNotIn

assertNotIn asserts that element **is** **not** present **in** the collection.

**try**:

tc.assertNotIn(a, [a])

**except** Exception **as** e:

print(e)

assertIsInstance

tc.assertIsInstance(tc, object)

**try**:

print(tc.assertIsInstance(a,b))

**except** Exception **as** e:

print(e)

isinstance() arg 2 must be a type or tuple of types

**try**:

print(tc.assertIsInstance(a,c))

**except** Exception **as** e:

print(e)

**try**:

print(tc.assertIsInstance(a,d))

**except** Exception **as** e:

print(e)

assertNotIsInstance

**try**:

print(tc.assertNotIsInstance(tc, object))

**except** Exception **as** e:

print(e)

**class** A:

**pass**

**class** B:

**pass**

**class** C(A):

**pass**

a **=** A()

b **=** B()

c **=** C()

print(tc.assertIsInstance(a,B))

*# print(tc.assertNotIsInstance(a, A))*

**---------------------------------------------------------------------------**

**AssertionError** Traceback (most recent call last)

**<ipython-input-117-9310e996d1ed>** in <module>

**----> 1** print**(**tc**.**assertIsInstance**(**a**,**B**))**

2 **# print(tc.assertNotIsInstance(a, A))**

**c:\users\asha.t\appdata\local\programs\python\python37-32\lib\unittest\case.py** in assertIsInstance**(self, obj, cls, msg)**

1261 **if** **not** isinstance**(**obj**,** cls**):**

1262 standardMsg **=** **'%s is not an instance of %r'** **%** **(**safe\_repr**(**obj**),** cls**)**

**-> 1263** self**.**fail**(**self**.**\_formatMessage**(**msg**,** standardMsg**))**

1264

1265 **def** assertNotIsInstance**(**self**,** obj**,** cls**,** msg**=None):**

**c:\users\asha.t\appdata\local\programs\python\python37-32\lib\unittest\case.py** in fail**(self, msg)**

678 **def** fail**(**self**,** msg**=None):**

679 **"""Fail immediately, with the given message."""**

**--> 680 raise** self**.**failureException**(**msg**)**

681

682 **def** assertFalse**(**self**,** expr**,** msg**=None):**

**AssertionError**: <\_\_main\_\_.A object at 0x03027C50> is not an instance of <class '\_\_main\_\_.B'>

print(tc.assertNotIsInstance(a,B))

print(type(b), type(a))

None

<class '\_\_main\_\_.B'> <class '\_\_main\_\_.A'>

expectedFailure

If we know that a test case will fail **and** that condition should be treated **as** success,

then expectedFailure decorator option can used **as** shown **in** the below example.

**%%**writefile unitesting.py

**from** num **import** multiply

**import** unittest

**class** TestUM(unittest.TestCase):

**def** test\_numbers\_3\_4(self):

self.assertEqual( multiply(3,4), 12)

**def** test\_strings\_a\_3(self):

self.assertEqual( multiply('a',3), 'aaa')

**def** test\_string\_a\_b(self):

self.assertFalse(multiply('a', 'b'))

@unittest.expectedFailure

**def** test\_fail(self):

self.assertEqual(1, 0, "broken")

**def** skipUnlessHasattr(obj, attr):

**if** hasattr(obj, attr):

**return** **lambda** func: func

**return** unittest.skip("{!r} doesn't have {!r}".format(obj, attr))

​

unittest.main()

Overwriting unitesting.py

**!**python unitesting.py **-**v

test\_fail (\_\_main\_\_.TestUM) ... expected failure

test\_numbers\_3\_4 (\_\_main\_\_.TestUM) ... ok

test\_string\_a\_b (\_\_main\_\_.TestUM) ... ERROR

test\_strings\_a\_3 (\_\_main\_\_.TestUM) ... ok

======================================================================

ERROR: test\_string\_a\_b (\_\_main\_\_.TestUM)

----------------------------------------------------------------------

Traceback (most recent call last):

File "unitesting.py", line 14, in test\_string\_a\_b

self.assertFalse(multiply('a', 'b'))

File "C:\Users\asha.t\AppData\Local\Programs\Python\Python37-32\Python - Training\15 - Unit testing\num.py", line 5, in multiply

return a\*b

TypeError: can't multiply sequence by non-int of type 'str'

----------------------------------------------------------------------

Ran 4 tests in 0.005s

FAILED (errors=1, expected failures=1)

*#instead of executing the testcases from the command line one by one we can use run method of unitest to run the tescases of the*

*#class being loaded*

​

suite **=** unittest.TestLoader().loadTestsFromModule(TestUM())

*# print(dir(suite))*

suite.addTest(unittest.TestLoader().loadTestsFromTestCase(TestUM))

unittest.TextTestRunner().run(suite)

x...x...

----------------------------------------------------------------------

Ran 8 tests in 0.020s

OK (expected failures=2)

Out[66]:

<unittest.runner.TextTestResult run=8 errors=0 failures=0>

​

​

*#programatically also we can select the tescases*

suite **=** unittest.TestLoader().loadTestsFromModule(TestUM())

print(dir(suite))

suite.addTest(unittest.TestLoader().loadTestsFromTestCase(TestUM))

unittest.TextTestRunner().run(suite)

x...x...

['\_\_call\_\_', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_addClassOrModuleLevelException', '\_cleanup', '\_get\_previous\_module', '\_handleClassSetUp', '\_handleModuleFixture', '\_handleModuleTearDown', '\_removeTestAtIndex', '\_removed\_tests', '\_tearDownPreviousClass', '\_tests', 'addTest', 'addTests', 'countTestCases', 'debug', 'run']

----------------------------------------------------------------------

Ran 8 tests in 0.018s

OK (expected failures=2)

Out[67]:

<unittest.runner.TextTestResult run=8 errors=0 failures=0>

**Skipping test cases**

We can use decorator unittest options skip, skipif or skipUnless to skip the testcases as shown in the below example. skip: Skips the testcase irrespective of condition. skipif: Skips the testcase if the condition is valid. skipUnless: Skips the testcase if the condition is not valid.

**%%**writefile skipfile.py

​

**import** unittest

**import** sys

**class** MyTestCase(unittest.TestCase):

​

version **=** 2

@unittest.skip("demonstrating skipping")

**def** test\_nothing(self):

self.fail("shouldn't happen")

**def** test\_something(self):

**pass**

​

@unittest.skipIf(version **<** 3,

"not supported in this library version")

**def** test\_format(self):

*# Tests that work for only a certain version of the library.*

**pass**

​

@unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")

**def** test\_windows\_support(self):

*# windows specific testing code*

**pass**

​

unittest.main()

Overwriting skipfile.py

suite **=** unittest.TestLoader().loadTestsFromModule(MyTestCase())

print(dir(suite))

suite.addTest(unittest.TestLoader().loadTestsFromTestCase(MyTestCase))

unittest.TextTestRunner().run(suite)

ss..ss..

['\_\_call\_\_', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_addClassOrModuleLevelException', '\_cleanup', '\_get\_previous\_module', '\_handleClassSetUp', '\_handleModuleFixture', '\_handleModuleTearDown', '\_removeTestAtIndex', '\_removed\_tests', '\_tearDownPreviousClass', '\_tests', 'addTest', 'addTests', 'countTestCases', 'debug', 'run']

----------------------------------------------------------------------

Ran 8 tests in 0.010s

OK (skipped=4)

Out[77]:

<unittest.runner.TextTestResult run=8 errors=0 failures=0>

**!** python skipfile.py **-**v

test\_format (\_\_main\_\_.MyTestCase) ... skipped 'not supported in this library version'

test\_nothing (\_\_main\_\_.MyTestCase) ... skipped 'demonstrating skipping'

test\_something (\_\_main\_\_.MyTestCase) ... ok

test\_windows\_support (\_\_main\_\_.MyTestCase) ... ok

----------------------------------------------------------------------

Ran 4 tests in 0.001s

OK (skipped=2)

​

​