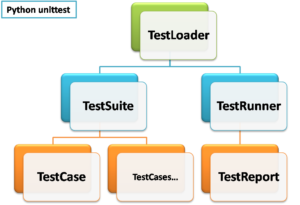
**Selenium Unit Testing in Python**

1. Understanding Python Unit test Framework And Its Features

* Python Unittest library inherits its root from a third-party module known as PyUnit.
* It was Steve Purcell who ideated PyUnit based on the famous JUnit framework.
* Like the JUnit, Python Unittest module splits up its functionality among five key components. All five elements work in tandem to support automation testing.

Five Components Of Python Unittest Framework



**Selenium Python Unittest Framework**

**Test Loader –** Python class which loads test cases and suites created locally or from an external data source like a file. It releases a TestSuite object that carries those cases and suites.

**Test Case –** The TestCase class holds the test handlers and provides hooks for preparing each handler and for cleaning up after execution.

**Test Suite –** It acts as a container for grouping test cases.

**Test Runner –** It provides a runnable interface for the execution of tests and delivers the results to the user. It can use channels like a GUI, a textual medium, or return a standard code to notify the results of test execution.

**Test Report –** This component organizes test results, display pass/fail status of the executed test cases. It even provides the details of steps, a summary of overall run and the time lapsed in execution.

Prepare A Test Case Class To Automate A Scenario

**assert functions :**

**a.** Use **assertEqual()** to check for an expected result.  
**b.** Use **assertTrue()** to verify a condition.  
**c.** Use **assertRaises()** to verify that an expected exception gets raised.

import unittest

from selenium import webdriver

class SearchText(unittest.TestCase):

Using SetUp() Method To Manage Test Pre-Requisites

A <setup()> method works as an entry point for the test cases. We can use it to run a fixed set of actions before executing a test or all the tests defined in the class.

**1.** Create an instance of a browser driver.  
**2.** Navigate to a base URL.  
**3.** Load tests data for execution.  
**4.** Open log files for recording inputs, statuses, and errors.

import unittest

from selenium import webdriver

class SearchText(unittest.TestCase):

def setUp(self):

# create a new Firefox session

self.driver = webdriver.Firefox()

self.driver.implicitly\_wait(30)

self.driver.maximize\_window()

# navigate to the application home page

self.driver.get("http://www.google.com/")

Start Writing Selenium Python Test Scripts Using Unittest

Create Selenium Test In Python With Unittest

**Use Case – In this example, we will search for a text in google and verify if the search returns a list of items**

import unittest

from selenium import webdriver

class SearchText(unittest.TestCase):

def setUp(self):

# create a new Firefox session

self.driver = webdriver.Firefox()

self.driver.implicitly\_wait(30)

self.driver.maximize\_window()

# navigate to the application home page

self.driver.get("http://www.google.com/")

def test\_search\_by\_text(self):

# get the search textbox

self.search\_field = self.driver.find\_element\_by\_name("q")

# enter search keyword and submit

self.search\_field.send\_keys("Selenium WebDriver Interview questions")

self.search\_field.submit()

#get the list of elements which are displayed after the search

#currently on result page usingfind\_elements\_by\_class\_namemethod

lists = self.driver.find\_elements\_by\_class\_name("r")

no=len(lists)

self.assertEqual(10, len(lists))

Define Cleanup Strategy To Free Resources Post Test Execution

Once the test execution finishes, the pre-requisites specified in the setup() method have to be cleaned up.

So to achieve this, the base TestCase class provides another method i.e. tearDown() which the runner calls after test execution. It lets us clean the values initialized at the beginning of the test via setup() method.

import unittest

from selenium import webdriver

class SearchText(unittest.TestCase):

def setUp(self):

# create a new Firefox session

self.driver = webdriver.Firefox()

self.driver.implicitly\_wait(30)

self.driver.maximize\_window()

# navigate to the application home page

self.driver.get("http://www.google.com/")

def test\_search\_by\_text(self):

# get the search textbox

self.search\_field = self.driver.find\_element\_by\_name("q")

# enter search keyword and submit

self.search\_field.send\_keys("Selenium WebDriver Interview questions")

self.search\_field.submit()

#get the list of elements which are displayed after the search

#currently on result page usingfind\_elements\_by\_class\_namemethod

lists = self.driver.find\_elements\_by\_class\_name("r")

no=len(lists)

self.assertEqual(11, len(lists))

def tearDown(self):

# close the browser window

self.driver.quit()

How To Execute The Test Case From Commandline?

Running the tests from the command line would require us to add a call to the main() method in the test script.

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

unittest.main()

After adding these lines, save the test as a standard Python script and name it as <**selenium-python-test.py**>. Then, try to execute it from the command line by using the following command.

python selenium-python-test.py

Add One More Selenium Python Test Case

def test\_search\_by\_name(self):

# get the search textbox

self.search\_field = self.driver.find\_element\_by\_name("q")

# enter search keyword and submit

self.search\_field.send\_keys("Python class")

self.search\_field.submit()

#get the list of elements which are displayed after the search

#currently on result page using find\_elements\_by\_class\_name method

list\_new = self.driver.find\_elements\_by\_class\_name("r")

self.assertEqual(10, len(list\_new))

Executing the TestClass would result in the first opening and then closing the two instances of Firefox. That’s how the setup() and tearDown() methods work for each test method. You can tally the results from the snapshot attached below.

2.5. Refactoring SetUp() And TearDown() Methods For Optimization

It was the **setup() method** which was causing this behavior as it runs before every test case. The Same case is with the **tearDown() method** which triggers for every test case after it finishes executing.

So we can refactor our script to minimize the resource usage. It means that we can share a single Firefox instance between the methods instead of creating a new instance every time.

It’s possible by using the **setUpClass()** and **tearDownClass()** methods along with the **@classmethod**decorator. These methods enable us to set the values at the class level rather than at the method level. The values initialized at class level are shared between the test methods.

the **setUpClass()** and **tearDownClass()** methods with the **@classmethod** decorator.

import unittest

from selenium import webdriver

class SearchText(unittest.TestCase):

@classmethod

def setUpClass(inst):

# create a new Firefox session

inst.driver = webdriver.Firefox()

inst.driver.implicitly\_wait(30)

inst.driver.maximize\_window()

# navigate to the application home page

inst.driver.get("http://www.google.com/")

inst.driver.title

def test\_search\_by\_text(self):

# get the search textbox

self.search\_field = self.driver.find\_element\_by\_name("q")

self.search\_field.clear()

# enter search keyword and submit

self.search\_field.send\_keys("Selenium Webdriver interview questions")

self.search\_field.submit()

#get the list of elements which are displayed after the search

#currently on result page using find\_elements\_by\_class\_name method

lists = self.driver.find\_elements\_by\_class\_name("r")

self.assertEqual(11, len(lists))

def test\_search\_by\_name(self):

# get the search textbox

self.search\_field = self.driver.find\_element\_by\_name("q")

# enter search keyword and submit

self.search\_field.send\_keys("Python class")

self.search\_field.submit()

#get the list of elements which are displayed after the search

#currently on result page using find\_elements\_by\_class\_name method

list\_new = self.driver.find\_elements\_by\_class\_name("r")

self.assertEqual(11, len(list\_new))

@classmethod

def tearDownClass(inst):

# close the browser window

inst.driver.quit()

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

unittest.main()

Upon executing the test, we can see that both the tests are getting run in the same Firefox browser.

3. Assertions In Python Unittest Framework

The TestCase class of the Python Unittest library implements a list of assert methods. We can use them to match actual values returned by the application with the expected values. With every method, we can specify a condition that must be true to continue executing the test.

Following three types of assert are available.

**1.** Checking equivalence.  
**2.** Logical comparison.  
**3.** Acting in the case of Exceptions.

While running a test, the execution moves to the next line only if the given assertion passes. Otherwise, the test would halt immediately prompting with a failure message.

List Of Assert Methods In Python Unittest Module

**assertEqual(a, b [,msg]) or assertNotEqual(a,b[,msg]) –**Check whether or not “a” and “b” match with each other. You can also pass a custom error message.

e.g. assertEqual(element.text,”10″)

**assertTrue(x[,msg])) or assertFalse(x[,msg])) or assertIsNot(a, b[,msg])) –**

Verify if the given expression evaluates to True or False.

e.g. assertTrue(element.is\_displayed())

**assertRaises(exc, fun, \*args, \*\*kwds) or assertRaisesRegexp(exc, r, fun, \*args, \*\*kwds) –**Check whether the test step raises the specific Exception mentioned. One such example is to use this method to check NoSuchElementFoundexception.

**assertAlmostEqual(a, b) or assertNotAlmostEqual(a,b) –** It compares the numeric values after rounding them to the number in the second argument.

**assertGreater(a, b) or assertGreaterEqual(a,b) or assertLess(a,b) or assertLessEqual(a,b) –** These methods are similar to the assertEqual() method.

**assertRegexpMatches(s, r) or assertNotRegexpMatches(s, r) –** Verify whether a regexpsearch matches the text.

**assertMultiLineEqual(a, b) –**This method is an extension to the assertEqual(), designed for multiline strings.

**assertListEqual(a, b) –**This method checks whether the lists “a” and “b” match. It helps to work with the drop-down fields.

**fail() –**This method fails the test unconditionally. It allows the creation of custom conditional blocks.

5. Generate HTML Test Suite Execution Report

By default, the Python Unittest library emits the test output on the terminal console. If you want to share the results with management and stakeholders, then sending console logs isn’t the appropriate way.

So you need to generate a report which looks presentable and professional. A summary report formatted nicely, with drill-down access to the details is what required.

Since the unit test library doesn’t have the ability to produce such a report, so you should use the HTMLTestRunner extension.

To download the HTMLTestRunner, please follow the below link.

[**+ DOWNLOAD HTMLTESTRUNNER MODULE**](http://tungwaiyip.info/software/HTMLTestRunner.html)

To integrate HTMLTestRunnersupport in our test script, we need to create an output file to store the actual report, configure the HTMLTestRunner options, and run the tests in the following manner.

Selenium Python Test Suite Example

import unittest

import HTMLTestRunner

import os

from SeleniumPythonRefactorTestCase import SearchText

from SeleniumPythonMultipleTests import HomePageTest

# get the directory path to output report file

dir = os.getcwd()

# get all tests from SearchText and HomePageTest class

search\_text = unittest.TestLoader().loadTestsFromTestCase(SearchText)

home\_page\_test = unittest.TestLoader().loadTestsFromTestCase(HomePageTest)

# create a test suite combining search\_text and home\_page\_test

test\_suite = unittest.TestSuite([home\_page\_test, search\_text])

# open the report file

outfile = open(dir + "\SeleniumPythonTestSummary.html", "w")

# configure HTMLTestRunner options

runner = HTMLTestRunner.HTMLTestRunner(stream=outfile,title='Test Report', description='Acceptance Tests')

# run the suite using HTMLTestRunner

runner.run(test\_suite)

After executing the test suite, we’ll see that HTMLTestRunner runs all the tests as does the test runner of the unit test library.