Unit Testing Python with Selemium

Let’s now look at the list of topics that we’re going to cover in this Selenium Python tutorial.

**1.**[**Understanding Python Unittest Framework and Its Features**](http://www.techbeamers.com/selenium-python-test-suite-unittest/#h1)**.**  
**1.1.**[**Five Components of Python Unittest Framework**](http://www.techbeamers.com/selenium-python-test-suite-unittest/#h1.1)**.**  
**1.2.**[**Prepare a Test Case Class to Automate a Scenario**](http://www.techbeamers.com/selenium-python-test-suite-unittest/#h1.2)**.**  
**1.3.**[**Using setUp() Method to Manage Test Pre-requisites**](http://www.techbeamers.com/selenium-python-test-suite-unittest/#h1.3)**.**  
**2.**[**Start Writing Selenium-Python Test Scripts Using Unittest**](http://www.techbeamers.com/selenium-python-test-suite-unittest/#h2)**.**  
**2.1.**[**Create Wer First Selenium Test in Python with Unittest**](http://www.techbeamers.com/selenium-python-test-suite-unittest/#h2.1)**.**  
**2.2.**[**Define Cleanup Strategy to Free Resources Post Test Execution**](http://www.techbeamers.com/selenium-python-test-suite-unittest/#h2.2)**.**  
**2.3.**[**How to Execute the Test Case from Commandline**](http://www.techbeamers.com/selenium-python-test-suite-unittest/#h2.3)**?**  
**2.4.**[**Add One More Selenium-Python Test Case**](http://www.techbeamers.com/selenium-python-test-suite-unittest/#h2.4)**.**  
**2.5.**[**Refactoring setUp() and tearDown() Methods for Optimization**](http://www.techbeamers.com/selenium-python-test-suite-unittest/#h2.5)**.**  
**3.**[**Assertions in Python Unittest Framework**](http://www.techbeamers.com/selenium-python-test-suite-unittest/#h3)**.  
3.1.**[**List of Assert Methods in Python Unittest Module**](http://www.techbeamers.com/selenium-python-test-suite-unittest/#h3.1)**.**  
**4.**[**Create Selenium-Python Test Suite Using Unittest**](http://www.techbeamers.com/selenium-python-test-suite-unittest/#h4)**.**  
**4.1.**[**Group Test Cases into a Test Suite**](http://www.techbeamers.com/selenium-python-test-suite-unittest/#h4.1)**.**  
**4.2.**[**Execute the Test Suite**](http://www.techbeamers.com/selenium-python-test-suite-unittest/#h4.2)**.**  
**5.**[**Generate HTML Test Suite Execution Report**](http://www.techbeamers.com/selenium-python-test-suite-unittest/#h5)**.**

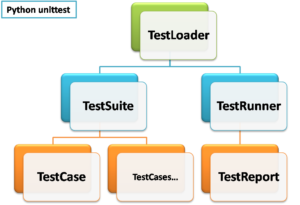
## Build A Selenium Python Test Suite From Scratch Using Unittest.

### 1. Understanding Python Unittest 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. And later it grew as an official Python module beginning from the version 2.5.

Like the JUnit, Python Unittest module splits up its functionality among five key components. All the five elements work in tandem to support automation testing. Let’s discuss each of them one by one in detail.

#### 1.1. Five Components Of Python Unittest Framework.



**Selenium Python Unittest Framework**

**Test Loader –** It’s a 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. With the help of a test suite, We can combine a set of test cases representing specific functionalities of the application under test.

**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, summary of overall run and the time lapsed in execution.

#### 1.2. Prepare A Test Case Class To Automate A Scenario.

We can create one or more tests by inheriting the TestCase class available in the Unittest module. To add a case, we also need to provide a corresponding test method (a handler) to the derived class. To finalize a test case, we can use assert or any of its variations for reporting test status.

Here are some of the most common assert functions used almost in all tests.

**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.

In addition to the test handler, we can also add routines like setUp() and tearDown() to manage creation and disposition of any objects or conditions that are mandatory for a test.

Let’s now start using the Unit test library and write a simple test by inheriting the TestCase class. For this, We’ll need to import the <unittest> module and define a class that inherits the TestCase class.

Look at the code below to get a feel of the test class.

import unittest

from selenium import webdriver

class SearchText(unittest.TestCase):

#### 1.3. 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.

These are pre-requisites which may include the following test setup preparation tasks.

**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.

This method takes no arguments and doesn’t return anything. If a script has the setUp() method defined, then the runner will call it first before running any of the test handlers.

In our example, we are using the setUp() method to create an instance of Firefox, setup the properties, and navigate to the main page of the application before executing the actual test.

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/")

### 2. Start Writing Selenium Python Test Scripts Using Unittest.

#### 2.1. Create Wer First Selenium Test In Python With Unittest.

After creating a setUp() method, we can now write some tests to verify the application’s functionality. So, first of all, let’s define our use case.

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

Similar to the <setUp()> method, test methods get implemented in the TestCase class. While adding these methods, it’s a good practice to prefix their names with the word test. Having such a name helps Test Runner distinguish between a test and other methods. Check out the below script to demonstrate the given Selenium Python example.

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))

#### 2.2. 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 test via setUp() method.

In our example, when the test execution ends, we no longer need the instance of Firefox. So we will close it in the tearDown() method, as shown in the following code.

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()

#### 2.3. 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. We’ll also pass a verbosity argument to the main(). It’ll get the test result details displayed on the console.

Below is the piece of code to facilitate command line execution. We’ll need to add it in our test script towards the end.

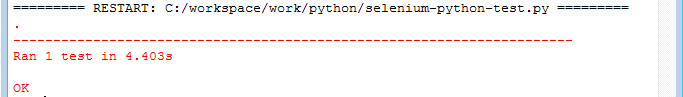
if \_\_name\_\_ == '\_\_main\_\_':

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.

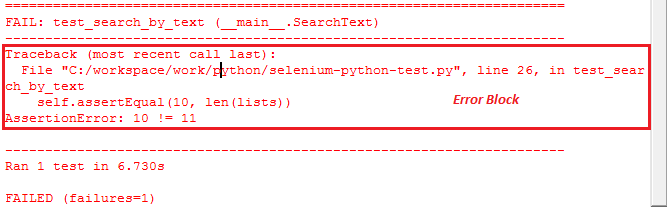
python selenium-python-test.py

After running the tests, the results would get displayed on the console along with the execution summary as captured in the following screenshot.



**First Selenium Python Test Case**

In addition to the results summary, there is a block of text getting printed as well to describe what went wrong. Look at the following screenshot to see what happens when we change the expected value (11 to 10) to something else.



**Print Error Text in Selenium Python Test Case**

As We can see from the logs, it’s easy to find the culprit test method that generated the failure. Use the backtrace to track down the code flow that led to the failure. Also, there is an AssertionError thrown after detecting a mismatch between expected and the actual output.

#### 2.4. Add One More Selenium Python Test Case.

So far, we’ve automated one simple test case. But we can add as many cases as expected in the TestCase class. It’ll also help in creating logical groups of tests that are related to a specific functionality. So let’s add another test to the TestCase class. Name the new method starting with the word test, as shown in the following code.

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()

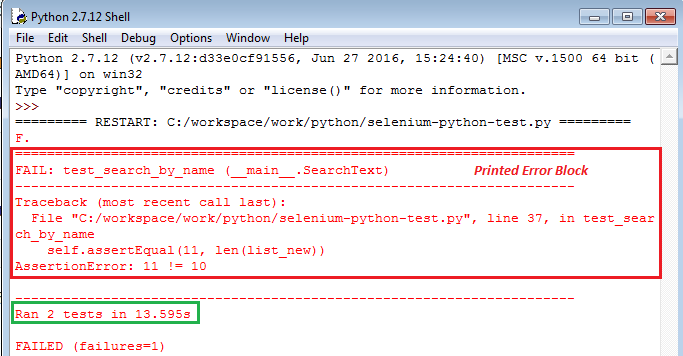
#get the list of elements which are displayed after the search

#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 first opening and then closing the two instances of Firefox. That’s how the setUp() and tearDown() methods work for each test method. We can tally the results from the snapshot attached below.



**Write Selenium Python Test Cases Using Unittest**

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

In the previous examples, we were using the setUp() method for creating instances of Firefox driver. But this approach was leading to the creation of a new instance of the web browser every time a new test case ran.

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 method level. The values initialized at class level are shared between the test methods.

Let’s see the above example with modified code to call the **setUpClass()** and **tearDownClass()** methods with the **@classmethod** decorator.

##### **Selenium Python Test Script Example.**

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 for 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 next line only if the given assertion passes. Otherwise, the test would halt immediately prompting with a failure message.

Let’s look at an important list of assert methods.

#### List Of Assert Methods In Python Unittest Module.

|  |  |  |
| --- | --- | --- |
| **Assert Method** | **Test Condition** | **Summary** |
| **assertEqual(a, b [,msg])** | a==b | Check whether or not “a” and “b” match with each other. We can also pass a cusotm error message.  e.g. assertEqual(element.text,”10″) |
| **assertNotEqual(a,b[,msg])** | a!=b |
| **assertTrue(x[,msg]))** | bool(x) is True | Verify if the given expression evaluates to True or False.  e.g. assertTrue(element.is\_displayed()) |
| **assertFalse(x[,msg]))** | bool(x) isFalse |
| **assertIsNot(a, b[,msg]))** | a is not b |
| **assertRaises(exc, fun,**  **\*args, \*\*kwds)** | fun(\*args,\*\*kwds)  raises exc | Check whether the test step raises the specific Exception mentioned. One such example is to use this method to check <NoSuchElementFoundexception>. |
| **assertRaisesRegexp(exc,**  **r, fun, \*args, \*\*kwds)** | fun(\*args,\*\*kwds)raises excand themessagematchesregex r |
| **assertAlmostEqual(a, b)** | round(a-b,7) == 0 | It compares the numeric values after rounding them to the number in the second argument. |
| **assertNotAlmostEqual(a,b)** | round(a-b,7) != 0 |
| **assertGreater(a, b)** | a > b | These methods are similar to the assertEqual() method. |
| **assertGreaterEqual(a,b)** | a>=b |
| **assertLess(a,b)** | a<b |
| **assertLessEqual(a,b)** | a<=b |
| **assertRegexpMatches(s, r)** | r.search(s) | Verify whether a regexpsearch matches the text. |
| **assertNotRegexpMatches(s, r)** | not r.search(s) |
| **assertMultiLineEqual(a,**  **b)** | strings | This method is an extension to the assertEqual(), designed for multilinestrings. |
| **assertListEqual(a, b)** | lists | 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. |

### 4. Create Selenium Python Test Suite Using Unittest.

The Unittest module has a TestSuite class which makes it easy to create an end to end Selenium Python Test Suite. With this feature, we can combine various tests into logical groups and turn them into a unified test suite. All of this is achievable by using the TestSuite, TestLoader, and TestRunner classes.

Before we get into details of TestSuite, let’s add a new test to check the home page of the application under test. We’ll aggregate this test along with the previous search tests into a single test suite, as shown in the following code.

#### Selenium Python Test Script Example.

import unittest

from selenium import webdriver

from selenium.common.exceptions import NoSuchElementException

from selenium.webdriver.common.by import By

class HomePageTest(unittest.TestCase):

@classmethod

def setUp(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/")

def test\_search\_box(self):

# check search box exists on Home page

self.assertTrue(self.is\_element\_present(By.NAME,"q"))

def test\_language\_settings(self):

# check language options on Home page

self.assertTrue(self.is\_element\_present(By.ID,"\_eEe"))

def test\_images\_link(self):

# check images link on Home page

images\_link = self.driver.find\_element\_by\_link\_text("Images")

images\_link.click()

# check search field exists on Images page

self.assertTrue(self.is\_element\_present(By.NAME,"q"))

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

# enter search keyword and submit

self.search\_field.send\_keys("Selenium Webdriver framework architecture diagram")

self.search\_field.submit()

@classmethod

def tearDown(inst):

# close the browser window

inst.driver.quit()

def is\_element\_present(self, how, what):

"""

Helper method to confirm the presence of an element on page

:params how: By locator type

:params what: locator value

"""

try: self.driver.find\_element(by=how, value=what)

except NoSuchElementException: return False

return True

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

unittest.main(verbosity=2)

#### 4.1. Group Test Cases Into A Test Suite.

We would have now very well understood that we’ll use the TestSuite class for defining and running the test suite. And we can add multiple test cases into it. Also, apart from the TestSuite class, we need to use TestLoader and TextTestRunner classes to create and run a test suite. Please refer the below code.

import unittest

from SeleniumPythonRefactorTestCase import SearchText

from SeleniumPythonMultipleTests import HomePageTest

# 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])

# run the suite

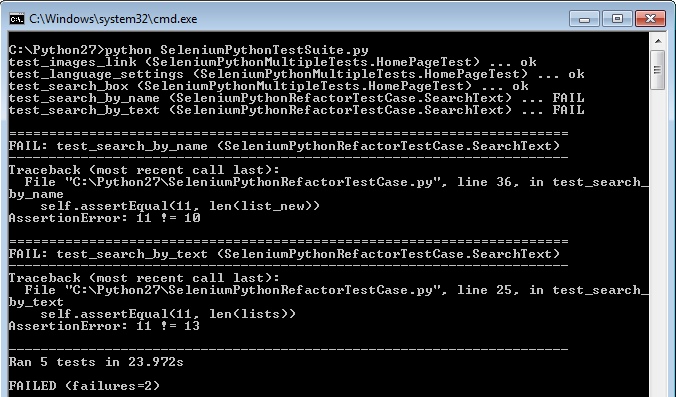
unittest.TextTestRunner(verbosity=2).run(test\_suite)

#### 4.2. Execute The Test Suite.

The TestLoader class reads all the test methods from the specified test files that contain the definition of the test suite. Then, the TestRunner class take control of the test suite and run all the tests specified. Below is the command to run the new test suite script.

python SeleniumPythonTestSuite.py

It’ll run all the tests from the SearchText and HomePage classes and generate the following output.



**First Selenium Python Test Case**

### 5. Generate HTML Test Suite Execution Report.

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

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

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

To download the HTMLTestRunner, please follow the below link.

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