Learning Selenium Testing Tools with Python

Selenium WebDriver is a popular automated testing tool for web applications. Python is one of the top programming languages and when used with Selenium it can automate and test web applications. Using Python’s unittest module, you can write test cases in Selenium. Over the years, Selenium has become a very powerful testing platform and many organizations are adopting Selenium WebDriver for creating automated user interface tests.

The book’s main aim is to cover the fundamentals related to Python Selenium testing. You will learn how the Selenium WebDriver Python API can be integrated with CI and Build tools to allow tests to be run while building applications. This book will guide you through using the Selenium WebDriver Python client library as well as other tools from the Selenium project. Towards the end of this book, you’ll get to grips with Selenium Grid, which is used for running tests in parallel using nodes for cross-browser testing. It will also give you a basic overview of the concepts, while helping you improve your practical testing skills with Python and Selenium.

Who this book is written for
If you are a quality testing professional, or a software or web application developer looking to create automation test scripts for your web applications, with an interest in Python, then this is the perfect guide for you. Python developers who need to do Selenium testing need not learn Java, as they can directly use Selenium for testing with this book.

What you will learn from this book
- Create Selenium WebDriver tests using the Python unittest module
- Use Selenium WebDriver for cross-browser testing
- Build reliable and robust tests using implicit and explicit waits
- Set up and use Selenium Grid for distributed run
- Test web applications on mobile platforms such as iOS and Android using Appium
- Utilize various methods provided by Selenium WebDriver to locate web elements and interact with them
- Capture screenshots and videos of the test execution
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Getting Started with Selenium WebDriver and Python'
- A synopsis of the book’s content
- More information on Learning Selenium Testing Tools with Python

 About the Author

Unmesh Gundecha has a Master’s degree in Software Engineering and over 12 years of experience in software development and testing. Unmesh has architected functional test automation solutions using industry standards, in-house and custom test automation frameworks along with leading commercial and open source test automation tools.

He has development experience in Microsoft technologies and Java. He is presently working as a test solution architect with a multinational company in Pune, India.

His experience includes support to Ruby, Java, iOS, Android, and PHP projects as an automated tester and software developer.


I would like to thank Neha Nagwekar, acquisition editor, for giving me an opportunity to write this book; Athira Laji, content development editor; Harshal Ved, project coordinator, for constantly supporting and encouraging me in completing this book; and the entire team at Packt Publishing.

This book has benefited a lot from a great team of technical reviewers. I'd like to thank each of them for volunteering their time reviewing the drafts of this book and providing valuable feedback.

I would also like to thank my mentor and friend, Yuri Weinstein, for his support and help during this project. Thanks to my friends and colleagues at work for their constant encouragement and support in writing this book.
Learning Selenium Testing Tools with Python

Selenium is a set of tools for automating browsers. It is largely used for testing applications, but its usages are not limited only to testing. It can also be used for screen scraping and automating repetitive tasks in a browser window. Selenium supports automation on all the major browsers including Firefox, Internet Explorer, Google Chrome, Safari, and Opera. Selenium WebDriver is now a part of W3C standards and is supported by major browser vendors.

Selenium offers the following set of tools for automating interaction with browsers:

- **Selenium IDE**: This is a Firefox add-in used to record and play back the Selenium scripts with Firefox. It provides a graphical user interface to record user actions using Firefox. It is a great tool to start learning and using Selenium, but it can only be used with Firefox and other browsers are not supported. However, it can convert the recorded scripts into various programming languages supported by Selenium WebDriver, which supports running scripts on browsers other than Firefox.

- **Selenium WebDriver**: This is a programming interface for developing advanced Selenium scripts using programming languages. We can also run tests on multiple browsers that are supported by Selenium on multiple operating systems, including Linux, Windows, and Mac OS X. This makes Selenium a true cross-browser testing tool. Selenium WebDriver offers client libraries in various languages, including Java, C#, Python, Ruby, PHP, and JavaScript, and are more into writing test scripts.

- **Selenium standalone server**: This is also known as Selenium Grid and allows remote and distributed execution of Selenium scripts created with WebDriver. We can also use the grid feature of the standalone server to run tests in parallel, including tests on mobile platforms such as Android or Apple iOS for iPhone and iPad.

As the title suggests, this book will introduce you to the Selenium WebDriver client library for Python. You will learn how to use Selenium WebDriver in Python to automate browsers for testing web applications. This book contains lessons right from setting up Selenium to using the basic and advanced features of Selenium to create and run automated scripts for testing web applications. This book assumes you have a basic idea of programming using Python.
What This Book Covers

Chapter 1, Getting Started with Selenium WebDriver and Python, starts with installing Python and the Selenium WebDriver client library. We will select a Python editor or IDE for Selenium script development. We will then create our first automation script for a simple search workflow from the application under test. At the end of this chapter, we will run the Selenium script on various browsers supported by Selenium.

Chapter 2, Writing Tests Using unittest, shows you how to use Selenium and the unittest library to test web applications. We will convert the script into a unittest test case. We will create few more tests using Selenium and unittest. We will create a TestSuite for a group of tests. We will run these tests and analyze the results. At the end of this chapter, you will learn how to produce test reports in the HTML format that you can distribute to various stakeholders of the project.

Chapter 3, Finding Elements, introduces you to locators that are the keys to automate different types of User Interface (UI) elements displayed on the web pages in the browser window. Selenium uses locators to find elements on a page and then performs actions or retrieves their properties for testing. You will learn various methods to locate elements, including XPath and CSS. We will show you how to use these methods with examples on the application under test.

Chapter 4, Using the Selenium Python API for Element Interaction, shows you how to use the Selenium WebDriver client library to interact with different types of elements, JavaScript alerts, frames, and windows in Python. You will learn how to perform actions such as sending values to elements, performing clicks, and selecting options from dropdowns. You will also see how to handle frames, different types of JavaScript alerts, and switch between child browser windows.

Chapter 5, Synchronizing Tests, introduces you to various wait methods provided by Selenium to synchronize tests for reliable and robust execution. You will learn how to use the implicit and explicit wait to implement synchronization in Selenium tests. You will learn various methods to implement explicit wait in our test scripts.

Chapter 6, Cross-browser Testing, dives into using RemoteWebDriver to run cross-browser tests on remote machines or through the Selenium Grid. You will learn how to use RemoteWebDriver to run tests on remote machines.

We will also set up a Selenium Grid to run tests on various combinations of browsers and operating systems. You will also see how to execute tests on headless browsers such as PhantomJS. At the end of the chapter, we will see how to use cloud testing tools such as Sauce Labs and BrowserStack to run tests in cloud using RemoteWebDriver.
Chapter 7, Testing on Mobile, shows you how to test applications on mobile devices using Selenium WebDriver and Appium. We will set up Appium to test our sample application on iOS and on an Android emulator and device. You will also learn how to run native mobile applications using Appium.

Chapter 8, Page Objects and Data-driven Testing, introduces you to two important design patterns to implement a maintainable and efficient testing framework. You will learn how to use page objects to hide the technical details of locators, and divide operations on pages into separate classes and create test cases that are more readable and easy to maintain. You will then learn how to create data-driven tests using the unittest library.

Chapter 9, Advanced Techniques of Selenium WebDriver, dives into some of the advanced techniques of using Selenium for automating browsers for testing. You will learn how to use various action methods for simulating complex mouse and keyboard operations using Selenium. You will see how to handle session cookies, capture screenshots during a test run, and create a movie of the entire test run.

Chapter 10, Integration with Other Tools and Frameworks, shows you how to use Selenium WebDriver with automated acceptance testing frameworks such as Behave and Continuous Integration tools. You will first learn how to integrate Selenium with Behave to create automated acceptance tests. We will implement a sample feature and acceptance tests on UI using the Selenium WebDriver. At end of the chapter, we will set up running the tests that we created as part of Continuous Integration using Jenkins. We will set up a schedule to run the tests on a daily frequency.

By the end of this book, you will have learned all the essential features of Selenium WebDriver to create your own web testing framework in Python.
Selenium automates browsers. It automates the interaction we do in a browser window such as navigating to a website, clicking on links, filling out forms, submitting forms, navigating through pages, and so on. It works on every major browser available out there.

In order to use Selenium WebDriver, we need a programming language to write automation scripts. The language that we select should also have a Selenium client library available.

In this book, we will use Python along with the Selenium WebDriver client library to create automated scripts. Python is a widely used general-purpose, high-level programming language. It's easy and its syntax allows us to express concepts in fewer lines of code. It emphasizes code readability and provides constructs that enable us to write programs on both the small and large scale. It also provides a number of in-built and user-written libraries to achieve complex tasks quite easily.

The Selenium WebDriver client library for Python provides access to all the Selenium WebDriver features and Selenium standalone server for remote and distributed testing of browser-based applications. Selenium Python language bindings are developed and maintained by David Burns, Adam Goucher, Maik Röder, Jason Huggins, Luke Semerau, Miki Tebeka, and Eric Allenin.

The Selenium WebDriver client library is supported on Python Version 2.6, 2.7, 3.2, and 3.3.

This chapter will introduce you to the Selenium WebDriver client library for Python by demonstrating its installation, basic features, and overall structure.
In this chapter, we will cover the following topics:

- Installing Python and Selenium package
- Selecting and setting up a Python editor
- Implementing a sample script using the Selenium WebDriver Python client library
- Implementing cross-browser support with Internet Explorer and Google Chrome

Preparing your machine

As a first step of using Selenium with Python, we'll need to install it on our computer with the minimum requirements possible. Let's set up the basic environment with the steps explained in the following sections.

Installing Python

You will find Python installed by default on most Linux distributions, Mac OS X, and other Unix machines. On Windows, you will need to install it separately. Installers for different platforms can be found at http://python.org/download/.

All the examples in this book are written and tested on Python 2.7 and Python 3.0 on Windows 8 operating systems.

Installing the Selenium package

The Selenium WebDriver Python client library is available in the Selenium package. To install the Selenium package in a simple way, use the pip installer tool available at https://pip.pypa.io/en/latest/.

With pip, you can simply install or upgrade the Selenium package using the following command:

```
pip install -U selenium
```

This is a fairly simple process. This command will set up the Selenium WebDriver client library on your machine with all modules and classes that we will need to create automated scripts using Python. The pip tool will download the latest version of the Selenium package and install it on your machine. The optional -U flag will upgrade the existing version of the installed package to the latest version.
You can also download the latest version of the Selenium package source from https://pypi.python.org/pypi/selenium. Just click on the `Download` button on the upper-right-hand side of the page, unarchive the downloaded file, and install it with following command:

```bash
git clone https://github.com/SeleniumHQ/selenium
```

Browsing the Selenium WebDriver Python documentation

The Selenium WebDriver Python client library documentation is available at http://selenium.googlecode.com/git/docs/api/py/api.html as shown in the following screenshot:
It offers detailed information on all core classes and functions of Selenium WebDriver. Also note the following links for Selenium documentation:

- The official documentation at http://docs.seleniumhq.org/docs/ offers documentation for all the Selenium components with examples in supported languages
- Selenium Wiki at https://code.google.com/p/selenium/wiki/lists some useful topics that we will explore later in this book

## Selecting an IDE

Now that we have Python and Selenium WebDriver set up, we will need an editor or an Integrated Development Environment (IDE) to write automation scripts. A good editor or IDE increases the productivity and helps in doing a lot of other things that make the coding experience simple and easy. While we can write Python code in simple editors such as Emacs, Vim, or Notepad, using an IDE will make life a lot easier. There are many IDEs to choose from. Generally, an IDE provides the following features to accelerate your development and coding time:

- A graphical code editor with code completion and IntelliSense
- A code explorer for functions and classes
- Syntax highlighting
- Project management
- Code templates
- Tools for unit testing and debugging
- Source control support

If you're new to Python, or you're a tester working for the first time in Python, your development team will help you to set up the right IDE.

However, if you're starting with Python for the first time and don't know which IDE to select, here are a few choices that you might want to consider.

### PyCharm

PyCharm is developed by JetBrains, a leading vendor of professional development tools and IDEs such as IntelliJ IDEA, RubyMine, PhpStorm, and TeamCity.

PyCharm is a polished, powerful, and versatile IDE that works pretty well. It brings best of the JetBrains experience in building powerful IDEs with lots of other features for a highly productive experience.
PyCharm is supported on Windows, Linux, and Mac. To know more about PyCharm and its features visit http://www.jetbrains.com/pycharm/.

PyCharm comes in two versions—a community edition and a professional edition. The community edition is free, whereas you have to pay for the professional edition. Here is the PyCharm community edition running a sample Selenium script in the following screenshot:
The community edition is great for building and running Selenium scripts with its fantastic debugging support. We will use PyCharm in the rest of this book. Later in this chapter, we will set up PyCharm and create our first Selenium script.

All the examples in this book are built using PyCharm; however, you can easily use these examples in your choice of editor or IDE.

The PyDev Eclipse plugin

The PyDev Eclipse plugin is another widely used editor among Python developers. Eclipse is a famous open source IDE primarily built for Java; however, it also offers support to various other programming languages and tools through its powerful plugin architecture.

Eclipse is a cross-platform IDE supported on Windows, Linux, and Mac. You can get the latest edition of Eclipse at http://www.eclipse.org/downloads/.

You need to install the PyDev plugin separately after setting up Eclipse. Use the tutorial from Lars Vogel to install PyDev at http://www.vogella.com/tutorials/Python/article.html to install PyDev. Installation instructions are also available at http://pydev.org/.
Here's the Eclipse PyDev plugin running a sample Selenium script as shown in the following screenshot:

PyScripter
For the Windows users, PyScripter can also be a great choice. It is open source, lightweight, and provides all the features that modern IDEs offer such as IntelliSense and code completion, testing, and debugging support. You can find more about PyScripter along with its download information at https://code.google.com/p/pyscripter/.
Here’s PyScripter running a sample Selenium script as shown in the following screenshot:
Chapter 1

Setting up PyCharm

Now that we have seen IDE choices, let’s set up PyCharm. All examples in this book are created with PyCharm. However, you can set up any other IDE of your choice and use examples as they are. We will set up PyCharm with following steps to get started with Selenium Python:

2. Launch the PyCharm Community Edition. Click on the Create New Project option on the PyCharm Community Edition dialog box as shown in the following screenshot:
3. On the Create New Project dialog box, as shown in next screenshot, specify the name of your project in the Project name field. In this example, setests is used as the project name. We need to configure the interpreter for the first time. Click on the button to set up the interpreter, as shown in the following screenshot:

![Create New Project dialog box](image)

4. On the Python Interpreter dialog box, click on the plus icon. PyCharm will suggest the installed interpreter similar to the following screenshot. Select the interpreter from Select Interpreter Path.

![Select Interpreter Path](image)
5. PyCharm will configure the selected interpreter as shown in the following screenshot. It will show a list of packages that are installed along with Python. Click on the Apply button and then on the OK button:

![Python Interpreters](image)

6. On the Create New Project dialog box, click on the OK button to create the project:

![Create New Project](image)
Taking your first steps with Selenium and Python

We are now ready to start with creating and running automated scripts in Python. Let's begin with Selenium WebDriver and create a Python script that uses Selenium WebDriver classes and functions to automate browser interaction.

We will use a sample web application for most of the examples in this book. This sample application is built on a famous e-commerce framework—Magento. You can find the application at http://demo.magentocommerce.com/.

In this sample script, we will navigate to the demo version of the application, search for products, and list the names of products from the search result page with the following steps:

1. Let's use the project that we created earlier while setting up PyCharm. Create a simple Python script that will use the Selenium WebDriver client library. In Project Explorer, right-click on setests and navigate to New | Python File from the pop-up menu:
2. On the New Python file dialog box, enter searchproducts in the Name field and click on the OK button:

![New Python file dialog box](image)

3. PyCharm will add a new tab searchproducts.py in the code editor area. Copy the following code in the searchproduct.py tab:

```python
from selenium import webdriver

# create a new Firefox session
driver = webdriver.Firefox()
driver.implicitly_wait(30)
driver.maximize_window()

# navigate to the application home page
driver.get("http://demo.magentocommerce.com/")

# get the search textbox
search_field = driver.find_element_by_name("q")
search_field.clear()

# enter search keyword and submit
search_field.send_keys("phones")
search_field.submit()

# get all the anchor elements which have product names displayed
# currently on result page using find_elements_by_xpath method
products = driver.find_elements_by_xpath("//h2[@class='product-name']/a")

# get the number of anchor elements found
print "Found " + str(len(products)) + " products:

# iterate through each anchor element and print the text that is
# name of the product
for product in products:
    print product.text

# close the browser window
driver.quit()
```
If you're using any other IDE or editor of your choice, create a new file, copy the code to the new file, and save the file as `searchproducts.py`.

4. To run the script, press the `Ctrl + Shift + F10` combination in the PyCharm code window or select Run 'searchproducts' from the Run menu. This will start the execution and you will see a new Firefox window navigating to the demo site and the Selenium commands getting executed in the Firefox window. If all goes well, at the end, the script will close the Firefox window. The script will print the list of products in the PyCharm console as shown in the following screenshot:
We can also run this script through the command line with the following command. Open the command line, then open the setests directory, and run following command:

```
python searchproducts.py
```

We will use command line as the preferred method in the rest of the book to execute the tests.

We'll spend some time looking into the script that we created just now. We will go through each statement and understand Selenium WebDriver in brief. There is a lot to go through in the rest of the book.

The `selenium.webdriver` module implements the browser driver classes that are supported by Selenium, including Firefox, Chrome, Internet Explorer, Safari, and various other browsers, and `RemoteWebDriver` to test on browsers that are hosted on remote machines.

We need to import `webdriver` from the Selenium package to use the Selenium WebDriver methods:

```python
from selenium import webdriver
```

Next, we need an instance of a browser that we want to use. This will provide a programmatic interface to interact with the browser using the Selenium commands. In this example, we are using Firefox. We can create an instance of Firefox as shown in following code:

```python
driver = webdriver.Firefox()
```

During the run, this will launch a new Firefox window. We also set a few options on the driver:

```python
driver.implicitly_wait(30)
driver.maximize_window()
```

We configured a timeout for Selenium to execute steps using an implicit wait of 30 seconds for the driver and maximized the Firefox window through the Selenium API. We will learn more about implicit wait in Chapter 5, Synchronizing Tests.

Next, we will navigate to the demo version of the application using its URL by calling the `driver.get()` method. After the `get()` method is called, WebDriver waits until the page is fully loaded in the Firefox window and returns the control to the script.
After loading the page, Selenium will interact with various elements on the page, like a human user. For example, on the Home page of the application, we need to enter a search term in a textbox and click on the **Search** button. These elements are implemented as HTML input elements and Selenium needs to find these elements to simulate the user action. Selenium WebDriver provides a number of methods to find these elements and interact with them to perform operations such as sending values, clicking buttons, selecting items in dropdowns, and so on. We will see more about this in Chapter 3, Finding Elements.

In this example, we are finding the **Search** textbox using the `find_element_by_name` method. This will return the first element matching the name attribute specified in the `find` method. The HTML elements are defined with tag and attributes. We can use this information to find an element, by following the given steps:

1. In this example, the **Search** textbox has the name attribute defined as `q` and we can use this attribute as shown in the following code example:

   ```python
   search_field = driver.find_element_by_name("q")
   ```

2. Once the **Search** textbox is found, we will interact with this element by clearing the previous value (if entered) using the `clear()` method and enter the specified new value using the `send_keys()` method. Next, we will submit the search request by calling the `submit()` method:

   ```python
   search_field.clear()
   search_field.send_keys("phones")
   search_field.submit()
   ```

3. After submission of the search request, Firefox will load the result page returned by the application. The result page has a list of products that match the search term, which is `phones`. We can read the list of results and specifically the names of all the products that are rendered in the anchor `<a>` element using the `find_elements_by_xpath()` method. This will return more than one matching element as a list:

   ```python
   products =
   driver.find_elements_by_xpath("//h2[@class='product-name']/a")
   ```

4. Next, we will print the number of products (that is the number of anchor `<a>` elements) that are found on the page and the names of the products using the `.text` property of all the anchor `<a>` elements:

   ```python
   print "Found " + str(len(products)) + " products:"
   for product in products:
       print product.text
   ```
5. At end of the script, we will close the Firefox browser using the `driver.quit()` method:

```python
driver.quit()
```

This example script gives us a concise example of using Selenium WebDriver and Python together to create a simple automation script. We are not testing anything in this script yet. Later in the book, we will extend this simple script into a set of tests and use various other libraries and features of Python.

**Cross-browser support**

So far we have built and run our script with Firefox. Selenium has extensive support for cross-browser testing where you can automate on all the major browsers including Internet Explorer, Google Chrome, Safari, Opera, and headless browsers such as PhantomJS. In this section, we will set up and run the script that we created in the previous section with Internet Explorer and Google Chrome to see the cross-browser capabilities of Selenium WebDriver.

**Setting up Internet Explorer**

There is a little more to run scripts on Internet Explorer. To run tests on Internet Explorer, we need to download and set up the `InternetExplorerDriver` server. The `InternetExplorerDriver` server is a standalone server executable that implements WebDriver's wire protocol to work as glue between the test script and Internet Explorer. It supports major IE versions on Windows XP, Vista, Windows 7, and Windows 8 operating systems. Let's set up the `InternetExplorerDriver` server with the following steps:

1. Download the `InternetExplorerDriver` server from [http://www.seleniumhq.org/download/](http://www.seleniumhq.org/download/). You can download 32- or 64-bit versions based on the system configuration that you are using.

2. After downloading the `InternetExplorerDriver` server, unzip and copy the file to the same directory where scripts are stored.

3. On IE 7 or higher, the Protected Mode settings for each zone must have the same value. Protected Mode can either be on or off, as long as it is for all the zones. To set the Protected Mode settings:
   1. Choose Internet Options from the Tools menu.
   2. On the Internet Options dialog box, click on the Security tab.
3. Select each zone listed in Select a zone to view or change security settings and make sure Enable Protected Mode (requires restarting Internet Explorer) is either checked or unchecked for all the zones. All the zones should have the same settings as shown in the following screenshot:

![Internet Options](image)

While using the InternetExplorerDriver server, it is also important to keep the browser zoom level set to 100 percent so that the native mouse events can be set to the correct coordinates.

4. Finally, modify the script to use Internet Explorer. Instead of creating an instance of the Firefox class, we will use the IE class in the following way:

```python
import os
from selenium import webdriver
```
# get the path of IEDriverServer
dir = os.path.dirname(__file__)
ie_driver_path = dir + "\IEDriverServer.exe"

# create a new Internet Explorer session
driver = webdriver.Ie(ie_driver_path)
driver.implicitly_wait(30)
driver.maximize_window()

# navigate to the application home page
driver.get("http://demo.magentocommerce.com/")

# get the search textbox
search_field = driver.find_element_by_name("q")
search_field.clear()

# enter search keyword and submit
search_field.send_keys("phones")
search_field.submit()

# get all the anchor elements which have product names displayed
# currently on result page using find_elements_by_xpath method
products = driver.find_elements_by_xpath("//h2[@class='product-name']/a")

# get the number of anchor elements found
print "Found " + str(len(products)) + " products:"

# iterate through each anchor element and print the text that is
# name of the product
for product in products:
    print product.text

# close the browser window
driver.quit()

In this script, we passed the path of the InternetExplorerDriver server
while creating the instance of an IE browser class.

5. Run the script and Selenium will first launch the InternetExplorerDriver server,
which launches the browser, and execute the steps.
The InternetExplorerDriver server acts as an intermediary between the
Selenium script and the browser. Execution of the actual steps is very similar
to what we observed with Firefox.

### Setting up Google Chrome

Setting up and running Selenium scripts on Google Chrome is similar to Internet Explorer. We need to download the ChromeDriver server similar to InternetExplorerDriver. The ChromeDriver server is a standalone server developed and maintained by the Chromium team. It implements WebDriver's wire protocol for automating Google Chrome. It is supported on Windows, Linux, and Mac operating systems. Set up the ChromeDriver server using the following steps:

2. After downloading the ChromeDriver server, unzip and copy the file to the same directory where the scripts are stored.
3. Finally, modify the sample script to use Chrome. Instead of creating an instance of the Firefox class, we will use the Chrome class in the following way:

   ```python
   import os
   from selenium import webdriver

   # get the path of chromedriver
dir = os.path.dirname(__file__)
   chrome_driver_path = dir + '\chromedriver.exe'
   # remove the .exe extension on linux or mac platform

   # create a new Chrome session
driver = webdriver.Chrome(chrome_driver_path)
driver.implicitly_wait(30)
driver.maximize_window()

   # navigate to the application home page
driver.get("http://demo.magentocommerce.com/")

   # get the search textbox
   search_field = driver.find_element_by_name("q")
   search_field.clear()

   # enter search keyword and submit
   ```
search_field.send_keys("phones")
search_field.submit()

# get all the anchor elements which have product names displayed
# currently on result page using find_elements_by_xpath method
products = driver.find_elements_by_xpath("//h2[@class='product-name']/a")

# get the number of anchor elements found
print "Found " + str(len(products)) + " products:

# iterate through each anchor element and print the text that is
# name of the product
for product in products:
    print product.text

# close the browser window
driver.quit()

In this script, we passed the path of the ChromeDriver server while creating an instance of the Chrome browser class.

4. Run the script. Selenium will first launch the Chromedriver server, which launches the Chrome browser, and execute the steps. Execution of the actual steps is very similar to what we observed with Firefox.


Summary

In this chapter, we introduced you to Selenium and its components. We installed the selenium package using the pip tool. Then we looked at various Editors and IDEs to ease our coding experience with Selenium and Python and set up PyCharm. Then we built a simple script on a sample application covering some of the high-level concepts of Selenium WebDriver Python client library using Firefox. We ran the script and analyzed the outcome. Finally, we explored the cross-browser testing support of Selenium WebDriver by configuring and running the script with Internet Explorer and Google Chrome.

In next chapter, we will learn how to use the unittest library to create automated tests using Selenium WebDriver. We will also learn how to create a suite of tests and run tests in groups.
Where to buy this book

You can buy Learning Selenium Testing Tools with Python from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.