This tutorial will make web UI testing easy. We will build a simple yet robust web UI test solution using [Python](https://blog.testproject.io/2019/05/16/python-testing-framework-pros-cons/), pytest, and [Selenium WebDriver](https://blog.testproject.io/2017/11/28/inside-selenium-webdriver/). We will learn strategies for good test design as well as patterns for good automation code. By the end of the tutorial, you’ll be a web [test automation](https://testproject.io/) champ! Your Python test project can be the foundation for your own test cases, too.

Tutorial Chapters

1. [Set Your Test Automation Goals](https://blog.testproject.io/2019/07/16/set-your-test-automation-goals/) (Chapter 1)
2. [Create A Python Test Automation Project Using Pytest](https://blog.testproject.io/2019/07/16/python-test-automation-project-using-pytest) (Chapter 2)
3. [Installing Selenium WebDriver Using Python and Chrome](https://blog.testproject.io/2019/07/16/installing-selenium-webdriver-using-python-chrome) (Chapter 3)
4. **You’re here →**[**Write Your First Web Test Using Selenium WebDriver, Python and Chrome**](https://blog.testproject.io/2019/07/16/web-test-using-selenium-webdriver-python-chrome)**(Chapter 4)**
5. [Develop Page Object Selenium Tests Using Python](https://blog.testproject.io/2019/07/16/develop-page-object-selenium-tests-using-python) (Chapter 5)
6. [How to Read Config Files in Python Selenium Tests](https://blog.testproject.io/2019/07/16/read-config-files-in-python-selenium) (Chapter 6)
7. [Take Your Python Test Automation To The Next Level](https://blog.testproject.io/2019/07/16/python-test-automation-next-level) (Chapter 7)
   * [Create Pytest HTML Test Reports](https://blog.testproject.io/2019/07/16/create-pytest-html-test-reports) (Chapter 7.1)
   * [Parallel Test Execution with Pytest](https://blog.testproject.io/2019/07/16/parallel-test-execution-with-pytest) (Chapter 7.2)
   * [Scale Your Test Automation using Selenium Grid and Remote WebDrivers](https://blog.testproject.io/2019/07/16/using-selenium-grid-and-remote-webdrivers) (Chapter 7.3)
   * [Test Automation for Mobile Apps using Appium and Python](https://blog.testproject.io/2019/07/16/test-automation-for-mobile-apps-using-appium-and-python) (Chapter 7.4)
   * [Create Behavior-Driven Python Tests using Pytest-BDD](https://blog.testproject.io/2019/07/16/behavior-driven-python-tests-using-pytest-bdd) (Chapter 7.5)

With WebDriver ready to go, let’s write our first web test! The test will be a simple DuckDuckGo search. [DuckDuckGo](https://duckduckgo.com/) is a search engine that doesn’t track user data. Users can enter search phrases and get links to matching websites, just like any other search engine.

Before writing [automation code](https://testproject.io/), it’s always best to write the test procedure in plain language. Writing the procedure forces us to think first and foremost about the behavior under test. Here’s our test procedure:

1. Navigate to the DuckDuckGo home page
2. Enter the search phrase
3. Verify that:
   1. Results appear on the results page
   2. The search phrase appears in the search bar
   3. At least one search result contains the search phrase

It’s fairly basic, but it covers typical searching behavior end-to-end.

The Code

Add the following test function to tests/test\_web.py:

1. **def** test\_basic\_duckduckgo\_search(browser):
2. URL = 'https://www.duckduckgo.com'
3. PHRASE = 'panda'
4. browser.get(URL)
5. search\_input = browser.find\_element\_by\_id('search\_form\_input\_homepage')
6. search\_input.send\_keys(PHRASE + Keys.RETURN)
7. link\_divs = browser.find\_elements\_by\_css\_selector('#links > div')
8. **assert** len(link\_divs) > 0
9. xpath = f"//div[@id='links']//\*[contains(text(), '{PHRASE}')]"
10. results = browser.find\_elements\_by\_xpath(xpath)
11. **assert** len(results) > 0
12. search\_input = browser.find\_element\_by\_id('search\_form\_input')
13. **assert** search\_input.get\_attribute('value') == PHRASE

The test\_basic\_duckduckgo\_search function implements our test procedure following the [Arrange-Act-Assert pattern](https://blog.testproject.io/2017/09/04/3-building-great-test-automation-suite/). Notice that the test function declares a parameter named browser, which is the same name as our fixture for ChromeDriver setup and cleanup. pytest will automatically call the fixture and inject the WebDriver reference whenever this test is run. The test function then uses the browser variable to make several WebDriver calls. Let’s see how those calls work.

Arrange

URL = 'https://www.duckduckgo.com'

The test declares the URL for the DuckDuckGo home page as a variable for readability and maintainability.

PHRASE = 'panda'

This is the search phrase that the test will use. Since the test covers a “basic” search, the phrase doesn’t matter too much. Other tests exercising different behaviors should use more complex phrases. Again, the test declares it at the top of the test function for readability and maintainability.

browser.get(URL)

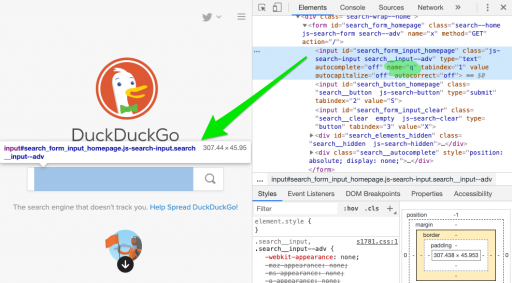
The starting point for the test is the DuckDuckGo home page. This call navigates the browser to the given URL. Be warned, though: this call does *not* wait for the page to load. It simply initiates the loading interaction.

Act

search\_input = browser.find\_element\_by\_id('search\_form\_input\_homepage')

The first step to automating web interactions is to find the target element(s). Elements may or may not appear on the page. Automation must use a [locator](https://automationpanda.com/2019/01/15/web-element-locators-for-test-automation/) to find the element if it does exist and then construct an object representing that element. There are many types of locators: [IDs](https://www.w3schools.com/html/html_id.asp), [class names](https://www.w3schools.com/html/html_classes.asp), [CSS selectors](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors), [XPaths](https://www.w3schools.com/xml/xml_xpath.asp), and more. [Locators](https://blog.testproject.io/2017/08/10/css-selector-custom-attributes/) will find all matching elements on the page – there could be more than one. Try to use the simplest locator that will uniquely identify the target element(s).

To write locators, you need to see the HTML structure of the page. [Chrome DevTools](https://developers.google.com/web/tools/chrome-devtools/) makes it easy to inspect the markup of any live page. Simply right-click the page and choose “Inspect”. You can see all elements on the “Elements” tab. For our test, we want to find the search input field on the DuckDuckGo home page. That element has an *id* attribute with the value “search\_form\_input\_homepage”, as shown below:



We can get that element using WebDriver’s find\_element\_by\_id method. The search\_input variable is assigned the object representing the search input element on the page. Remember, since the WebDriver instance has an implicit wait, it will wait up to 10 seconds for the search input element to appear on the page.

search\_input.send\_keys(PHRASE + Keys.RETURN)

With the element in hand, we can trigger interactions with it. The send\_keys method sends a sequence of keystrokes to the search input element, just like a human user would type at a keyboard. The call above sends the search phrase. The RETURN key at the end submits the search.

Assert (1)

link\_divs = browser.find\_elements\_by\_css\_selector('#links > div')

The result page should display a div element with an ID named “links” that has a child div element for each result link. The CSS selector above finds all such result link divs. Notice that “elements” is plural – this call will return a list.

**assert** len(link\_divs) > 0

The test must verify that results did actually appear for the search phrase. This assert statement makes sure that at least one result link was found on the page.

Assert (2)

xpath = f"//div[@id='links']//\*[contains(text(), '{PHRASE}')]"

Verifying that some results appeared is good, but we should also verify that the results match our search phrase. We can use an XPath to pinpoint result links that contain our search phrase in the text. XPaths are more complex than names and CSS selectors, but they are also more powerful. The XPath above searches for any div with the ID of “links”, and then it looks for descendants that contain the search phrase text.

(Does the “f” before the string look new to you? That’s an [f-string](https://realpython.com/python-f-strings/), and it makes string formatting easy!)

phrase\_results = browser.find\_elements\_by\_xpath(xpath)

This call finds all elements using the previously concatenated XPath. We could have combined these two lines into one, but splitting the lines is more readable and more Pythonic.

**assert** len(phrase\_results) > 0

Like the previous assertion, this one makes sure that at least one element was found. It’s a simple sanity check. It could be made to be more robust – like verifying that every result on the page contains the search phrase text – but that would be hard. Not every result may contain the exact text of the search phrase. For example, some might have uppercase characters. Locators and logic would need to be much more complicated for an advanced verification. Since this is a *basic* search test, the simpler assertion should be sufficient.

Assert (3)

search\_input = browser.find\_element\_by\_id('search\_form\_input')

The final assertion verifies that the search phrase still appears in the search input. The line above is identical to the find element call from the *Arrange* phase. It finds the search input element again. Why can’t we just use the search\_input object again? Unfortunately, the previous element has gone *stale*. The page changed from the search page to the results page. Even though the element looks the same, it is different, and it also needs a new locator. Therefore, we need to fetch it freshly.

**assert** search\_input.get\_attribute('value') == PHRASE

The text typed into an input element is accessible as its “value” attribute. This line asserts that the “value” attribute is equal to the search phrase. It verifies that the phrase hasn’t disappeared.

Review and Run the Web Test

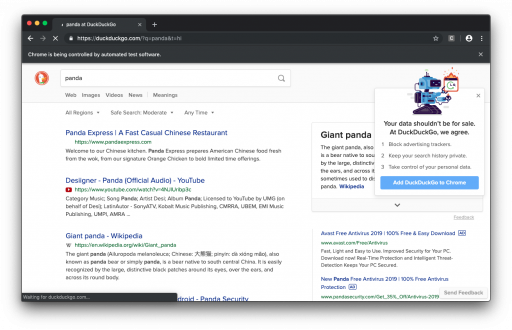
The full code for tests/test\_web.py should now look like this (with additional comments for clarity):

1. """
2. This module contains web test cases for the tutorial.
3. Tests use Selenium WebDriver with Chrome and ChromeDriver.
4. The fixtures set up and clean up the ChromeDriver instance.
5. """
6. **import** pytest
7. **from** selenium.webdriver **import** Chrome
8. **from** selenium.webdriver.common.keys **import** Keys
9. @pytest.fixture
10. **def** browser():
11. # Initialize ChromeDriver
12. driver = Chrome()
13. # Wait implicitly for elements to be ready before attempting interactions
14. driver.implicitly\_wait(10)
15. # Return the driver object at the end of setup
16. **yield** driver
17. # For cleanup, quit the driver
18. driver.quit()
19. **def** test\_basic\_duckduckgo\_search(browser):
20. # Set up some test case data
21. URL = 'https://www.duckduckgo.com'
22. PHRASE = 'panda'
23. # Navigate to the DuckDuckGo home page
24. browser.get(URL)
25. # Find the search input element
26. # In the DOM, it has an 'id' attribute of 'search\_form\_input\_homepage'
27. search\_input = browser.find\_element\_by\_id('search\_form\_input\_homepage')
28. # Send a search phrase to the input and hit the RETURN key
29. search\_input.send\_keys(PHRASE + Keys.RETURN)
30. # Verify that results appear on the results page
31. link\_divs = browser.find\_elements\_by\_css\_selector('#links > div')
32. **assert** len(link\_divs) > 0
33. # Verify that at least one search result contains the search phrase
34. xpath = f"//div[@id='links']//\*[contains(text(), '{PHRASE}')]"
35. phrase\_results = browser.find\_elements\_by\_xpath(xpath)
36. **assert** len(phrase\_results) > 0
37. # Verify that the search phrase is the same
38. search\_input = browser.find\_element\_by\_id('search\_form\_input')
39. **assert** search\_input.get\_attribute('value') == PHRASE

Now, run the test to make sure it works:

1. $ pipenv run python -m pytest
2. ============================= test session starts ==============================
3. platform darwin -- Python 3.7.3, pytest-4.5.0, py-1.8.0, pluggy-0.12.0
4. rootdir: /Users/andylpk247/Programming/automation-panda/python-webui-testing
5. collected 9 items
6. tests/test\_math.py ........ [ 88%]
7. tests/test\_web.py . [100%]
8. =========================== 9 passed **in** 6.10 seconds ===========================

When the web test runs, it will open Google Chrome. You can watch it automatically enter the search phrase, wait for the results page, and then quit the browser. Neat!



If the test fails to run, check the following:

* Does the test machine have Chrome installed?
* Is ChromeDriver on the system path?
* Does the ChromeDriver version match the Chrome version?
* Are there any filesystem permission problems?
* Are firewalls blocking any ports?
* Is the test code correct?