

Hands-On UI Testing with Python

Meet Your Presenters



Andy Knight



Nick Brown

2009

founded

11

global offices

6.5M+

users

22,000+

customers

11

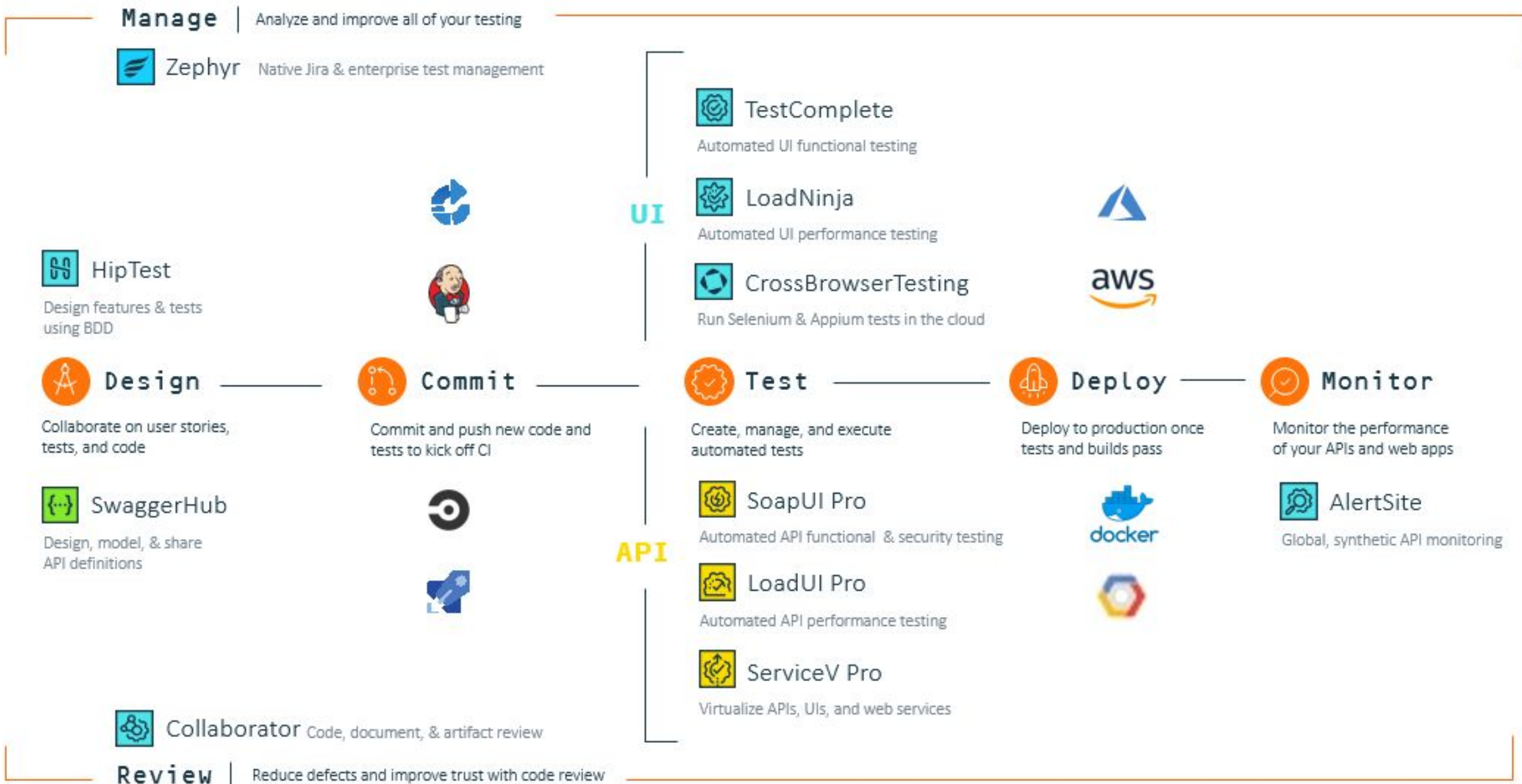
open source tools

525+

employees



SMARTBEAR



Web UI testing can be hard.
Let's make it easy.
Python can help.

Agenda

1. What is Web UI Testing?
2. Writing Our First Test
3. Making Browser Interactions
4. Testing Multiple Browsers
5. Live Q&A

GitHub Example Code:

AndyLPK247/smartbear-hands-on-ui-testing-python

Web UI Testing Overview

What is Web UI Testing?

Web UI testing is black box testing of a Web app through a browser.

- It is **feature testing** because it tests the app like a user.
- It is **end-to-end** because all parts are exercised together.

Since Web UI testing is expensive, good tests focus on ROI.

- Unit and integration tests should cover lower-level code.
- Web UI tests should focus on important, individual behaviors.
- Be careful about race conditions and changing pages!

Solution Sketch

Language

Python

Core Framework

pytest

UI Interactions

Page Object Pattern

Browser Automation

Selenium WebDriver

Multi-Browser Testing

CrossBrowserTesting

Solution Diagram

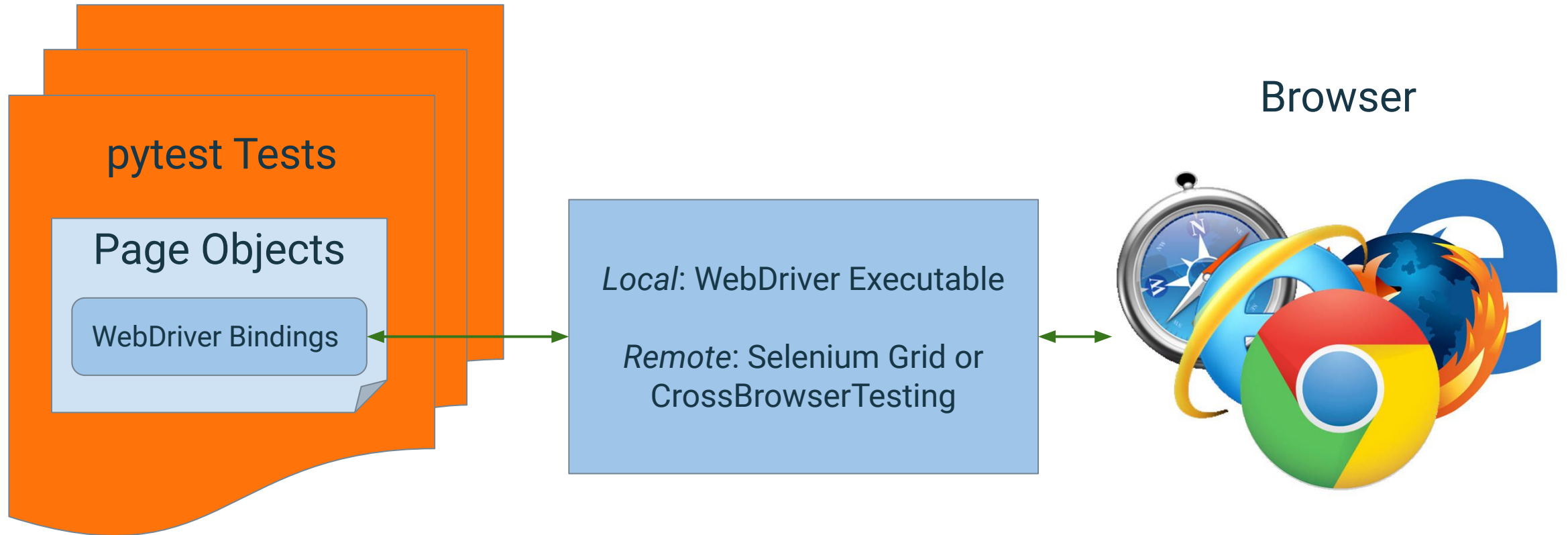


Image Source:

<https://www.zdnet.com/article/which-browser-is-most-popular-on-each-major-operating-system/>

Writing Our First Test

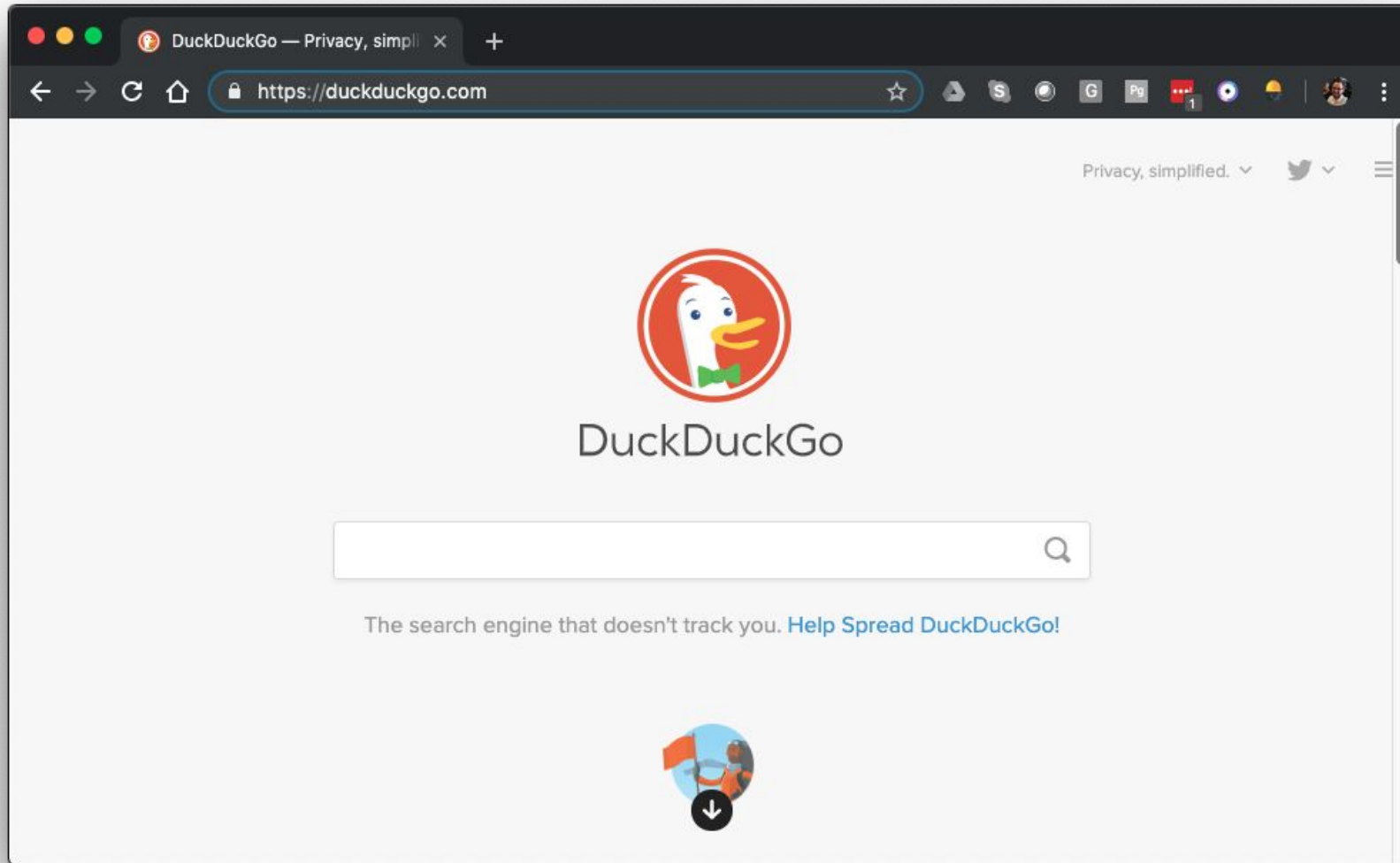
Our Web App to Test



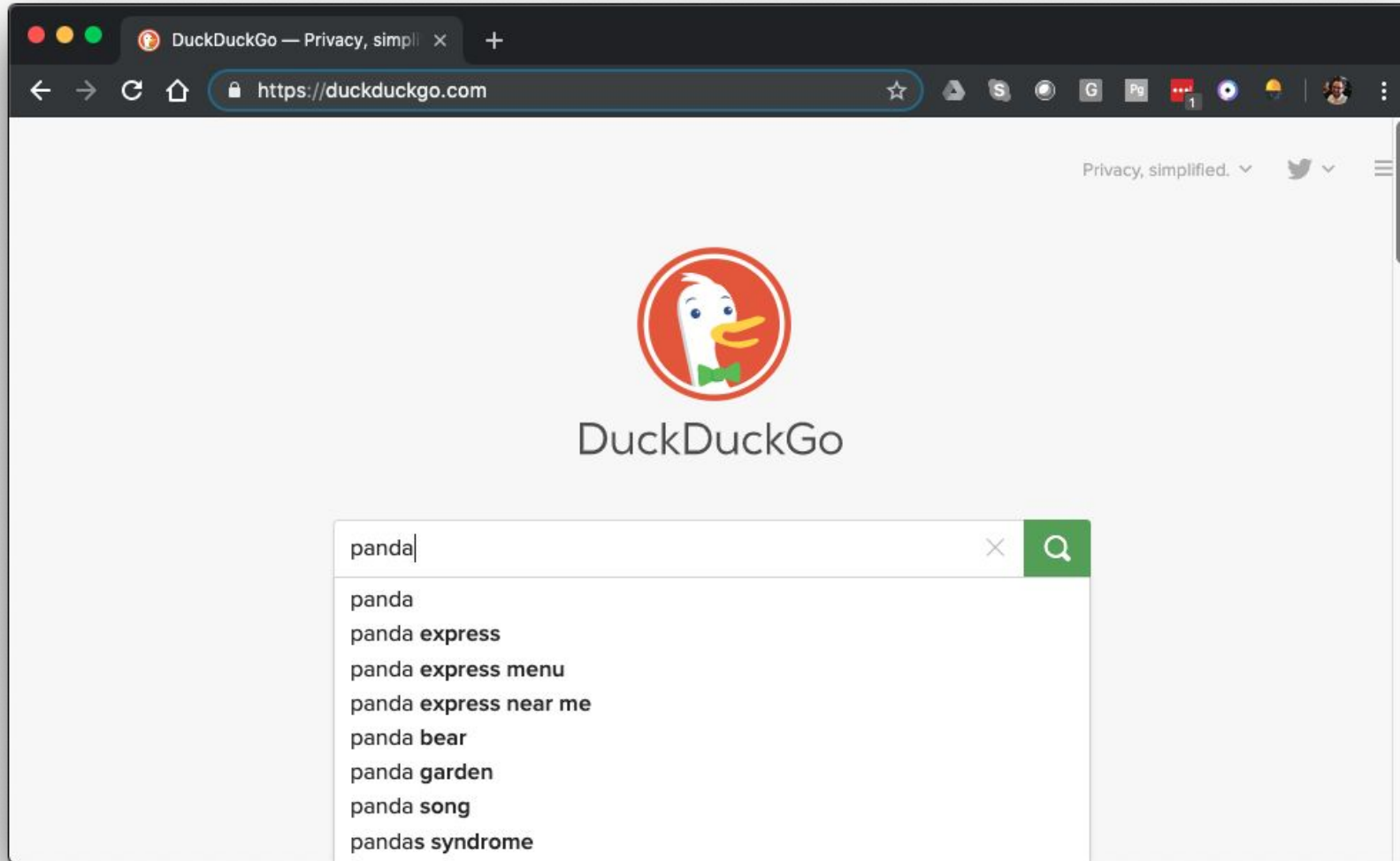
Let's write a basic search test together!

Always write test steps *before* test code.

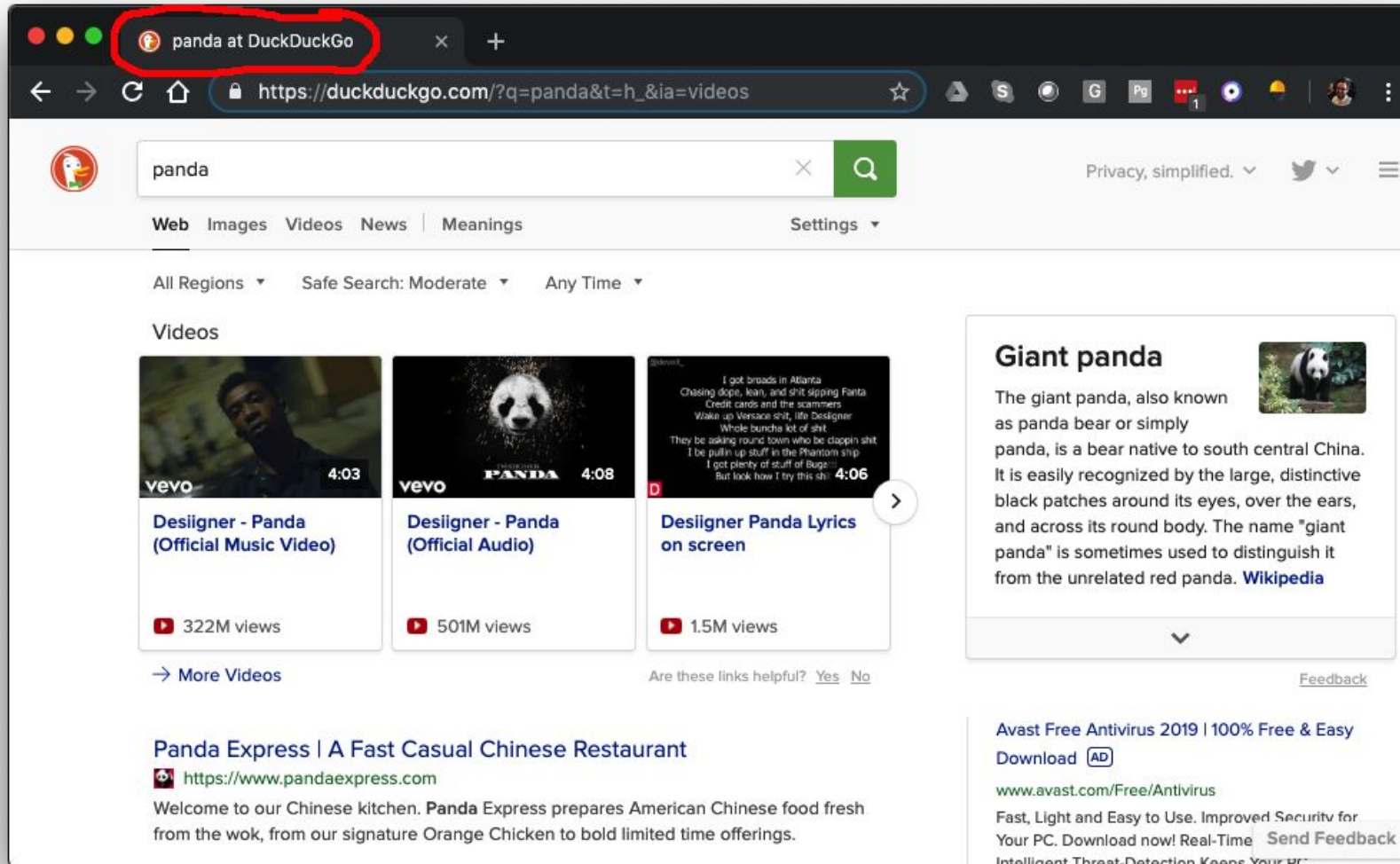
Step 1: Navigate to DuckDuckGo



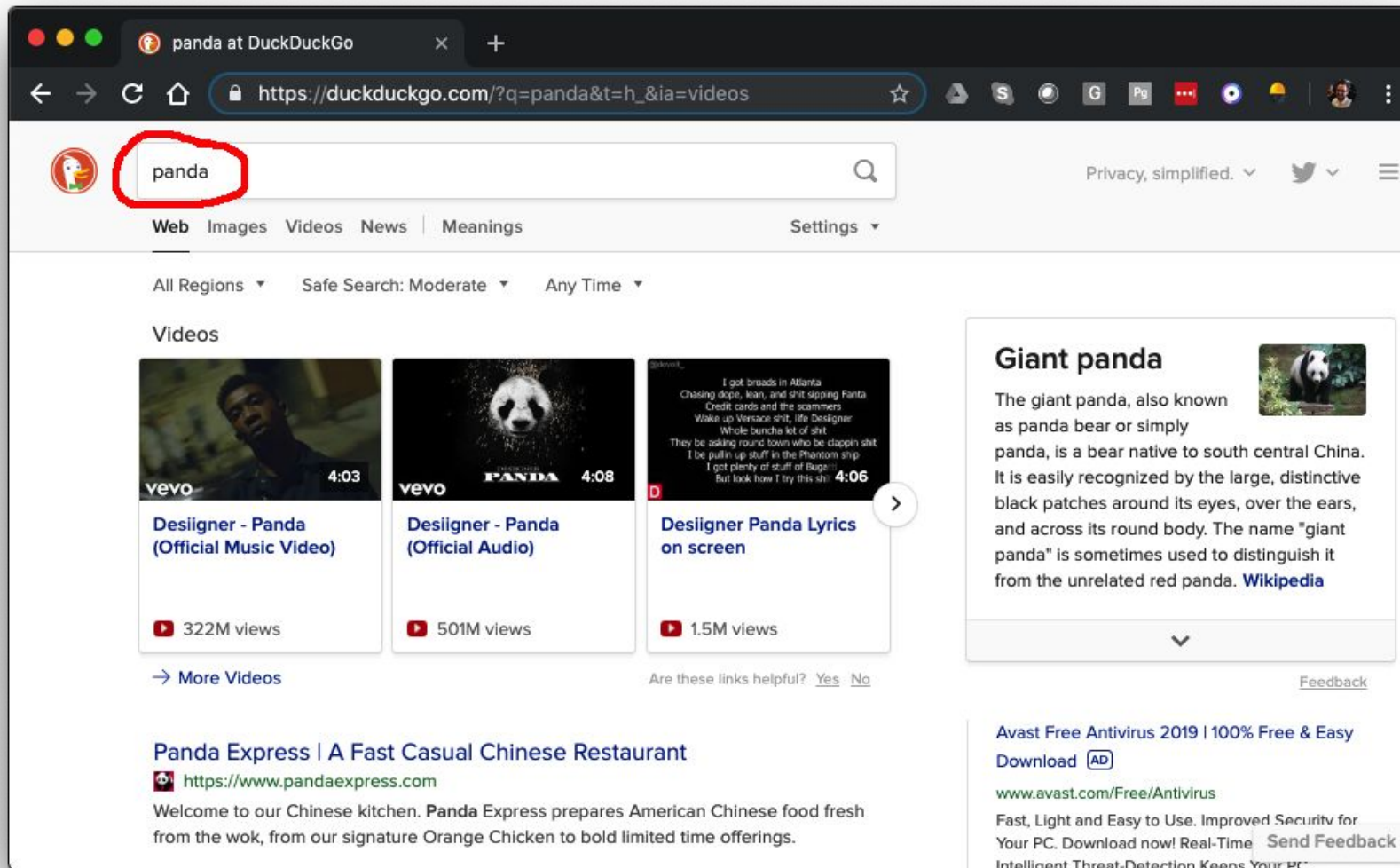
Step 2: Enter a search phrase



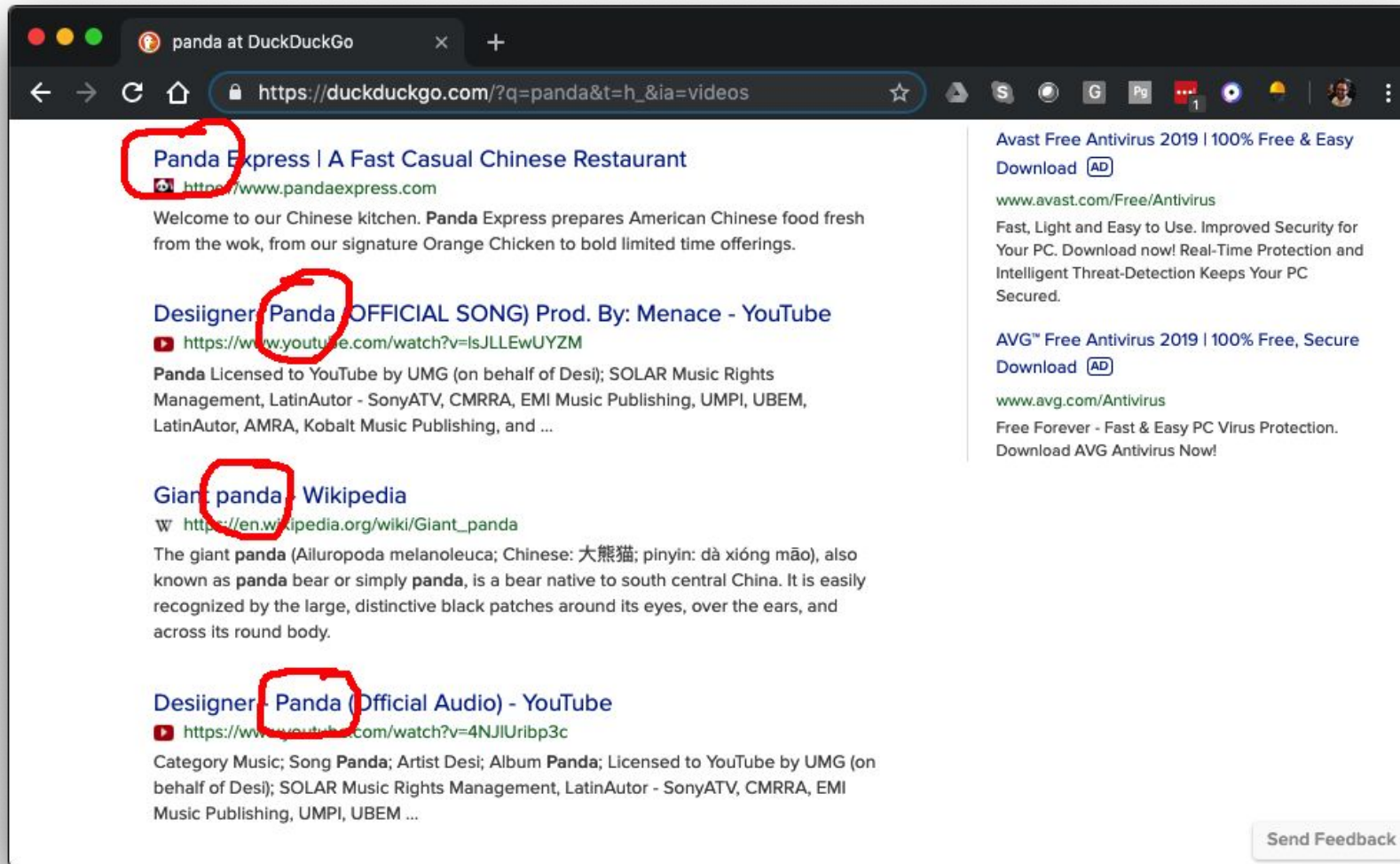
Step 3: Verify query in title



Step 4: Verify query on results page



Step 5: Verify results match query



Our First Test Case

Scenario: Basic DuckDuckGo Search

Given the DuckDuckGo home page is displayed

When the user searches for “panda”

Then the search result title contains “panda”

And the search result query is “panda”

And the search result links pertain to “panda”

Let's put this test into pytest.



About pytest

pytest is a mature full-featured Python testing tool that helps you write better programs.

pytest: helps you write better programs

The `pytest` framework makes it easy to write small tests, yet scales to support complex functional testing for applications and libraries.

An example of a simple test:

```
# content of test_sample.py
def inc(x):
    return x + 1

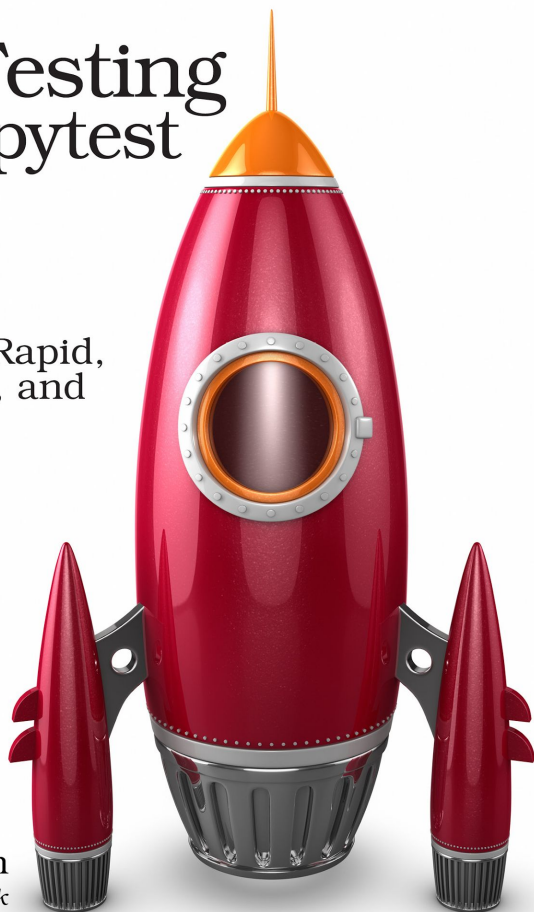
def test_answer():
    assert inc(3) == 5
```

The
Pragmatic
Programmers

Python Testing with pytest

Simple, Rapid,
Effective, and
Scalable

Brian Okken
edited by Katharine Dvorak



pytest Quick Start Guide

Write better Python code with simple and maintainable tests



By Bruno Oliveira

Packt>
www.packt.com

```
pip install pytest
```


Our First Test in Comments

```
def test_basic_duckduckgo_search():  
  
    # Given the DuckDuckGo home page is displayed  
    # TODO  
  
    # When the user searches for "panda"  
    # TODO  
  
    # Then the search result title contains "panda"  
    # TODO  
  
    # And the search result query is "panda"  
    # TODO  
  
    # And the search result links pertain to "panda"  
    # TODO  
  
    raise Exception("Incomplete Test")
```

Project Tree:

```
.  
└─ tests  
    └─ test_search.py
```

Making Browser Interactions

Selenium WebDriver

The **selenium** package is Selenium WebDriver for Python.


It sends Web UI commands from test automation code to a browser.

WebDriver can handle *every* type of Web UI interaction.

The best practice is to make all WebDriver calls from page objects.

7. WebDriver API — Selenium P x +

← → ↻ 🏠 🔒 https://selenium-python.readthedocs.io/api.html ☆ 🗑 ⚙ S 🌐 G Pg ⋮ 🎮 🔔 👤 🌙



7. WebDriver API

Note:

This is not an official documentation. Official API documentation is available [here](#).

Navigation

- 1. Installation
- 2. Getting Started
- 3. Navigating
- 4. Locating Elements
- 5. Waits
- 6. Page Objects
- 7. WebDriver API
 - 7.1. Exceptions
 - 7.2. Action Chains
 - 7.3. Alerts
 - 7.4. Special Keys
 - 7.5. Locate elements By
 - 7.6. Desired Capabilities
 - 7.7. Touch Actions
 - 7.8. Proxy
 - 7.9. Utilities
 - 7.10. Service
 - 7.11. Application Cache
 - 7.12. Firefox WebDriver
 - 7.13. Firefox WebDriver Options
 - 7.14. Firefox WebDriver

This chapter covers all the interfaces of Selenium WebDriver.

Recommended Import Style

The API definitions in this chapter show the absolute location of classes. However, the recommended import style is as given below:

```
from selenium import webdriver
```

Then, you can access the classes like this:

```
webdriver.Firefox
webdriver.FirefoxProfile
webdriver.Chrome
webdriver.ChromeOptions
webdriver.Ie
webdriver.Opera
webdriver.PhantomJS
webdriver.Remote
webdriver.DesiredCapabilities
webdriver.ActionChains
webdriver.TouchActions
webdriver.Proxy
```

The special keys class (Keys) can be imported like this:

```
from selenium.webdriver.common.keys import Keys
```

📄 v: latest ▾

```
pip install selenium
```

WebDriver setup?

Each test should have its own
WebDriver instance.

WebDriver Fixture

```
import pytest
import selenium.webdriver
```

```
@pytest.fixture
def browser():
    # This browser will be local
    # ChromeDriver must be on the system PATH
    b = selenium.webdriver.Chrome()
    b.implicitly_wait(10)
    yield b
    b.quit()
```

Project Tree:

```
.
├── tests
│   ├── conftest.py
│   └── test_search.py
```

The Page Object Pattern

A **page object** is an object representing a Web page or component.

- It has *locators* for finding elements on the page.
- It has *interaction methods* that interact with the page under test.

Each Web page or component under test should have a page object class.

- Page objects encapsulate low-level Selenium WebDriver calls.
- Tests can make short, readable calls instead of complicated ones.

Our Pages Under Test

DuckDuckGo Search Page

- Load the page
- Search a phrase

DuckDuckGo Result Page

- Get the result count
- Get the search query
- Get the title

The Search Page

```
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
```

```
class DuckDuckGoSearchPage:
```

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

```
    SEARCH_INPUT = (By.NAME, 'q')
```

```
    def __init__(self, browser):
        self.browser = browser
```

```
    def load(self):
        self.browser.get(self.URL)
```

```
    def search(self, phrase):
        search_input = self.browser.find_element(*self.SEARCH_INPUT)
        search_input.send_keys(phrase + Keys.RETURN)
```

Project Tree:

```
.
├── pages
│   ├── __init__.py
│   └── search.py
└── tests
    ├── conftest.py
    └── test_search.py
```

Locators

Locators are queries that find elements on a page.

There are many types:

- By.ID
- By.NAME
- By.CLASS_NAME
- By.CSS_SELECTOR
- By.XPATH
- By.LINK_TEXT
- By.PARTIAL_LINK_TEXT
- By.TAG_NAME

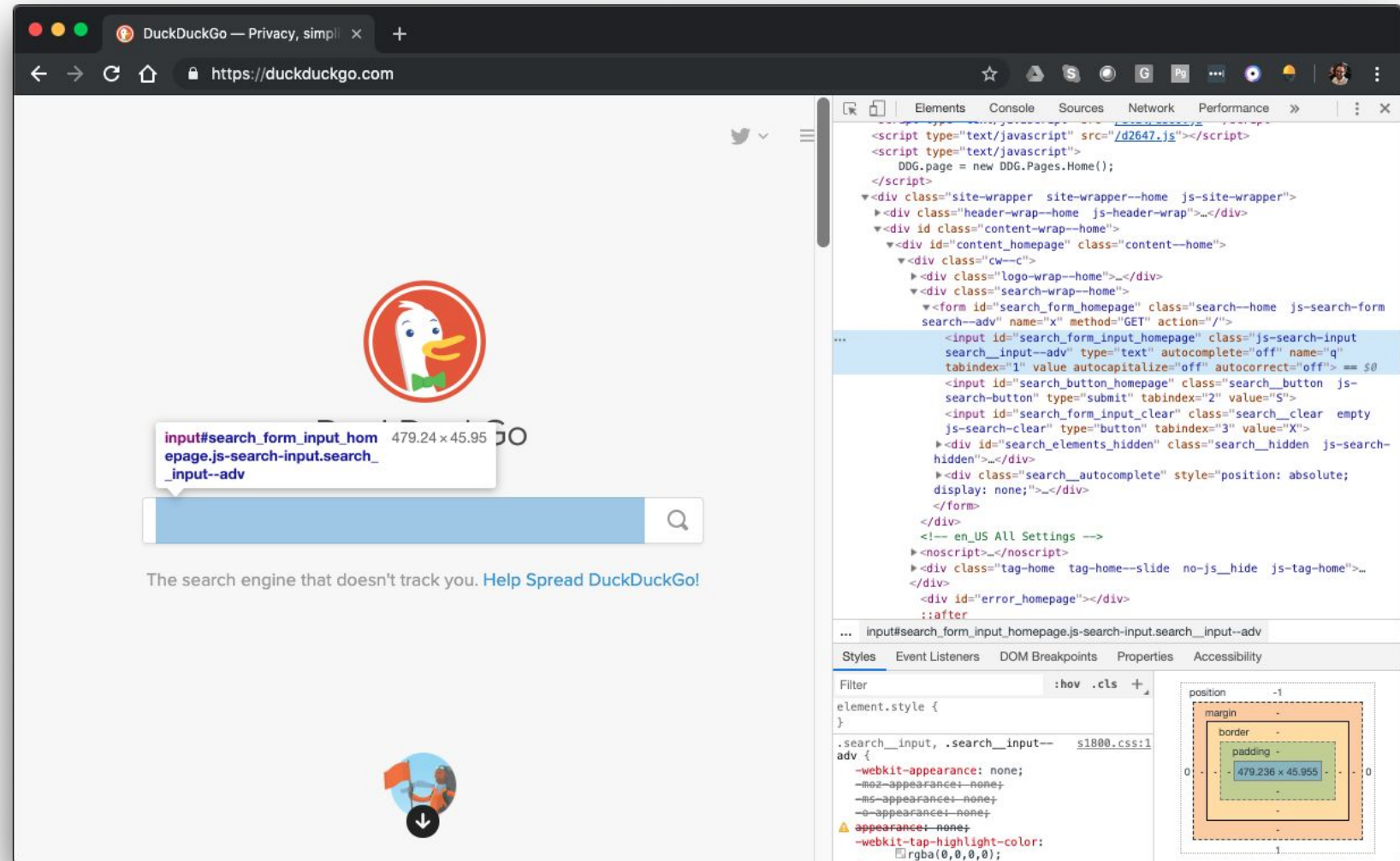
Want to learn more?
Take a free course online!

Test Automation University:
Web Element Locator Strategies

Finding Elements to Write Locators

Use
Chrome
DevTools!

Learn more
from TAU!



Add Page Object Calls to the Test

```
from pages.result import DuckDuckGoResultPage
from pages.search import DuckDuckGoSearchPage

def test_basic_duckduckgo_search(browser):
    search_page = DuckDuckGoSearchPage(browser)
    result_page = DuckDuckGoResultPage(browser)

    # Given the DuckDuckGo home page is displayed
    search_page.load()

    # When the user searches for "panda"
    search_page.search("panda")

    # Then the search result title contains "panda"
    assert "panda" in result_page.title()
    # And the search result query is "panda"
    assert "panda" == result_page.search_input_value()
    # And the search result links pertain to "panda"
    assert result_page.result_count_for_phrase("panda") > 0
```

Project Tree:

```
.
├── pages
│   ├── __init__.py
│   ├── result.py
│   └── search.py
└── tests
    ├── conftest.py
    └── test_search.py
```

A Successful Test Run

```
pyohio-2019-web-ui-testing — -bash — 80x24
[sterling2:pyohio-2019-web-ui-testing andylpk247$ pipenv run python -m pytest ]
===== test session starts =====
platform darwin -- Python 3.7.3, pytest-5.0.1, py-1.8.0, pluggy-0.12.0
rootdir: /Users/andylpk247/Programming/automation-panda/pyohio-2019-web-ui-testi
ng
collected 1 item

tests/test_search.py . [100%]

===== 1 passed in 5.16 seconds =====
sterling2:pyohio-2019-web-ui-testing andylpk247$
```

Testing Multiple Browsers

Testing Any Browser, Platform, and Version

Our current solution runs browser tests on the local machine.

Unfortunately, that doesn't scale well.

With **CrossBrowserTesting**, we can use any browser on any platform!

There's no need to set up our own Selenium Grid or device farm.

We can mitigate risk with a matrix of test configurations.

Testing in Parallel

We can also parallelize our tests using **pytest-xdist** with `CrossBrowserTesting`.

- *pytest-xdist* can launch multiple tests in parallel on the test machine.
- *CrossBrowserTesting* can handle multiple parallel sessions in the cloud.

Parallel testing can tremendously speed up time spent testing!

Config Data

```
{
  "authentication": {
    "username": "",
    "key": ""
  },
  "webdriver": {
    "name": "Hands-On UI Testing with Python",
    "build": "1.0",
    "browserName": "Chrome",
    "version": "75x64",
    "platform": "Windows 10",
    "screenResolution": "1366x768"
  }
}
```

Project Tree:

```
.
├── pages
│   ├── __init__.py
│   ├── result.py
│   └── search.py
├── tests
│   ├── conftest.py
│   └── test_search.py
└── cbt_config.json
```

Reading the Config Data

```
@pytest.fixture
def cbt_config(scope='session'):

    # Read the config file
    with open('cbt_config.json') as config_file:
        config = json.load(config_file)

    # Verify config
    assert 'authentication' in config
    assert 'username' in config['authentication']
    assert 'key' in config['authentication']
    assert 'webdriver' in config
    assert 'name' in config['webdriver']
    assert 'browserName' in config['webdriver']
    assert 'platform' in config['webdriver']

    # Return the config data
    return config
```

Project Tree:

```
.
├── pages
│   ├── __init__.py
│   ├── result.py
│   └── search.py
├── tests
│   ├── conftest.py
│   └── test_search.py
└── cbt_config.json
```

Using a Remote Browser

```
@pytest.fixture
def browser(cbt_config, request):

    # Concatenate the URL
    username = cbt_config['authentication']['username'].replace('@', '%40')
    key = cbt_config['authentication']['key']
    url = f"http://{username}:{key}@hub.crossbowtesting.com:80/wd/hub"

    # Request a remote browser from CrossBrowserTesting
    caps = cbt_config['webdriver']
    caps['name'] += ' | ' + request.node.name
    b = selenium.webdriver.Remote(
        desired_capabilities=caps, command_executor=url)

    # Increase the wait time, but all other code is the same
    b.implicitly_wait(30)
    yield b
    b.quit()
```

Project Tree:

```
.
├── pages
│   ├── __init__.py
│   ├── result.py
│   └── search.py
├── tests
│   ├── conftest.py
│   └── test_search.py
└── cbt_config.json
```

CrossBrowserTesting Local Connection: OFF Help 100% 100% 100% 100%

Test Center Live Testing Screenshots Automation

Automation Results Quick Start: Getting Started Legacy View Usage Details

Hands-On UI Testing with Python | test_basic_duckduckgo_search

10 75 1366x768 2 min ago

Group by Build

Hands-On UI Testing with Python | test_basic_duckduckgo_search

10 75 Running

Pass

Notes and Tags

Type notes here. Create tags by adding # before the word.

Save Notes

Commands

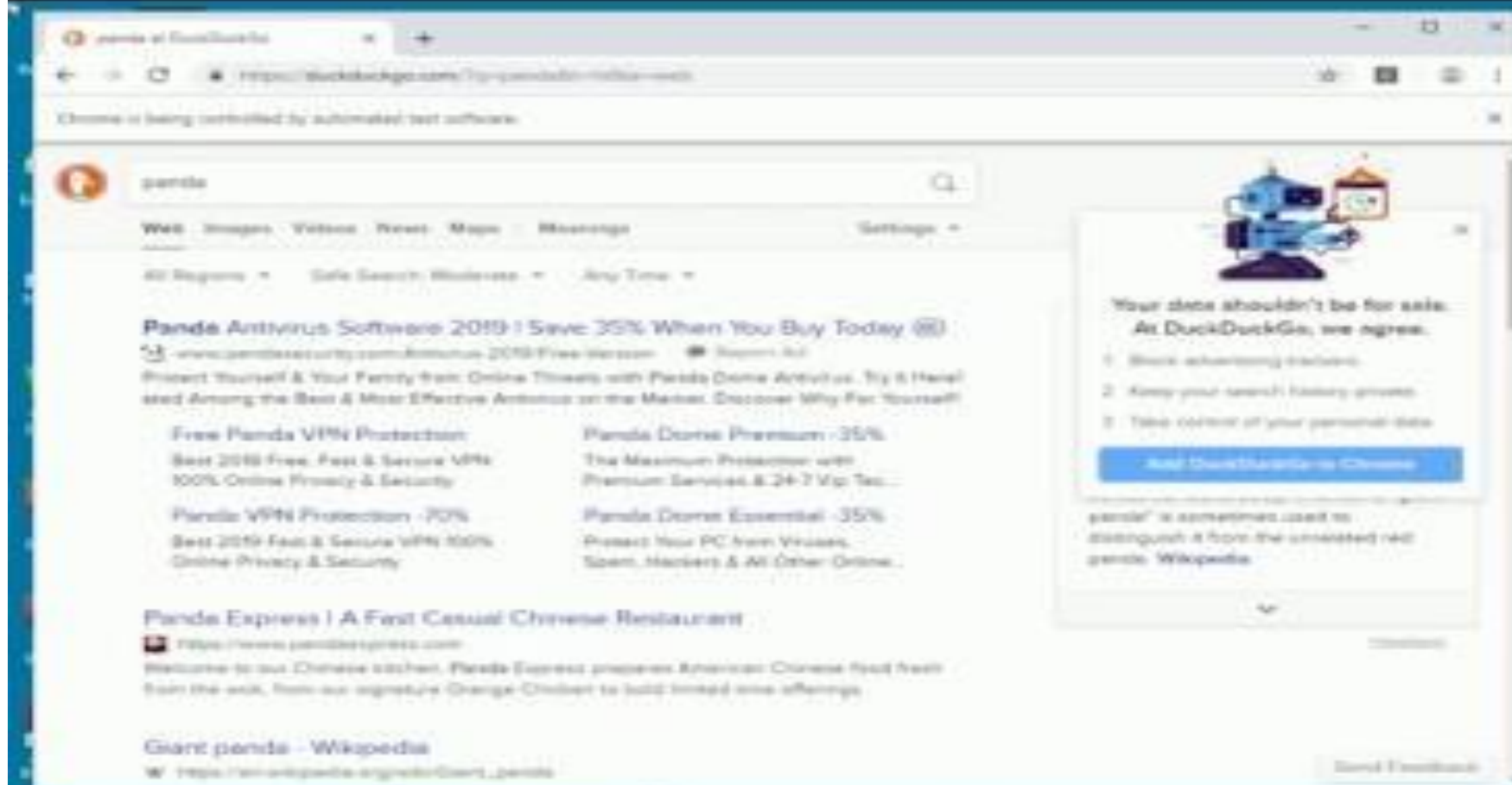
Video Snapshots: 0 Network Results Details

Raw

0:00 / 0:05

```
smartbear-hands-on-ui-testing-python -- python -m pytest -- BS+10
[sterling@smartbear-hands-on-ui-testing-python andylp247s ts
LICENSE Pipfile.lock xbt_config.json tests
Pipfile README.md pages
[sterling@smartbear-hands-on-ui-testing-python andylp247s pipenv run python -m pytest
===== test session starts =====
platform darwin -- Python 3.7.3, pytest-5.4.1, py-1.8.0, pluggy-0.12.0
rootdir: /Users/andylp247/Programming/automation-panda/smartbear-hands-on-ui-testing-py
then
collected 1 item

tests/test_search.py
```



Resources

- Example Code and Tutorials
 - This webinar: <https://github.com/AndyLPK247/smartbear-hands-on-ui-testing-python>
 - PyOhio 2019: <https://github.com/AndyLPK247/pyohio-2019-web-ui-testing>
- Test Automation University
 - Web Element Locator Strategies
 - Behavior-Driven Development with pytest-bdd
- Automation Panda blog
 - Testing page
 - Python page

Live Q&A

Thank You!

If you have any questions, feel free to reach out!

Email: Nicholas.Brown@smartbear.com