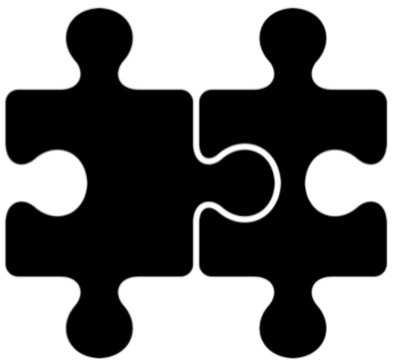
SELENIUM CHEAT SHEETS

Python Edition



by Dave Haeffner

Version 4.0.0

Table of Contents

Chapter 1
Local Configuration

Firefox

Option 1

1. Download the latest geckodriver binary
2. Add its location to your system path
3. Create an instance

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

Option 2

1. Download the latest geckodriver binary
2. Create an instance while specifying the path to the driver

```
driver = webdriver.Firefox(executable_path="/path/to/geckodriver")
```

NOTE: For more information about geckodriver check out its project page

Chrome

Option 1

1. Download the latest ChromeDriver
2. Add its location to your system path
3. Create an instance

```
driver = webdriver.Chrome()
```

Option 2

1. Download the latest ChromeDriver
2. Create an instance while specifying the path to the driver

```
driver = webdriver.Chrome("/path/to/chromedriver")
```

NOTE: For more information about ChromeDriver check out its project page

Internet Explorer

Option 1

1. Download the latest IEDriverServer.exe
2. Add its location to your path
3. Create an instance

```
webdriver.Ie()
```

Option 2

1. Download the latest IEDriverServer.exe
2. Create an instance while specifying the path to the driver

```
webdriver.Ie("/path/to/IEDriverServer.exe")
```

NOTE: There is additional setup required to make Internet Explorer work with Selenium. For more information check out the Selenium project Wiki page for InternetExplorerDriver.

Edge

In order to use Microsoft Edge you need to have access to Windows 10.

Option 1

1. Download EdgeDriver
2. Add its location to your path
3. Create an instance

```
driver = webdriver.Edge()
```

Option 2

1. Download EdgeDriver
2. Create an instance while specifying the path to the driver

```
driver = webdriver.Edge(executable_path="/path/to/edgedriver")
```

NOTE: You can download a free virtual machine with Windows 10 from Microsoft's Modern.IE developer portal. After that you need to download the appropriate `Microsoft WebDriver` server for your build of Windows. To find that go to `Start`, `Settings`, `System`, `About` and locate the number next to `OS Build` on the screen.

Safari

For Safari, you'll need SafariDriver, which ships with the latest version of Safari.

You just need to enable it from the command-line.

```
> safaridriver --enable
```

```
driver = webdriver.Safari()
```

NOTE: For additional details, or information on setup requirements for older versions of macOS, see the SafariDriver documentation from Apple.

Chapter 2
Cloud Configuration

Sauce Labs

Initial Setup

1. Create run-time flags with sensible defaults that can be overridden
2. Specify the browser and operating system you want through Desired Capabilitaies
3. Connect to Sauce Labs' end-point through the Desired Capabilities
4. Store the WebDriver instance returned for use in your tests

```
# conftest.py
def pytest_addoption(parser):
    parser.addoption("--baseurl",
                     action="store",
                     default="http://the-internet.herokuapp.com",
                     help="base URL for the application under test")
    parser.addoption("--host",
                     action="store",
                     default="saucelabs",
                     help="where to run your tests: localhost or saucelabs")
    parser.addoption("--browser",
                     action="store",
                     default="internet explorer",
                     help="the name of the browser you want to test with")
    parser.addoption("--browserversion",
                     action="store",
                     default="10.0",
                     help="the browser version you want to test with")
    parser.addoption("--platform",
                     action="store",
                     default="Windows 7",
                     help="the operating system to run your tests on (saucelabs only)")
```

```
# config.py
baseurl = ""
host = ""
browser = ""
browserversion = ""
platform = ""
```

```python
# conftest.py
import config
config.baseurl = request.config.getoption("--baseurl")
config.host = request.config.getoption("--host").lower()
config.browser = request.config.getoption("--browser").lower()
config.browserversion = request.config.getoption("--browserversion").lower()
config.platform = request.config.getoption("--platform").lower()

_desired_caps = {}
_desired_caps["browserName"] = config.browser
_desired_caps["version"] = config.browserversion
_desired_caps["platform"] = config.platform
_credentials = os.environ["SAUCE_USERNAME"] + ":" + os.environ["SAUCE_ACCESS_KEY"]
_url = "http://" + _credentials + "@ondemand.saucelabs.com:80/wd/hub"
driver_ = webdriver.Remote(_url, _desired_caps)
```

For more info:

- Sauce Labs Available Platforms page
- Sauce Labs Automated Test Configurator

Setting the Test Name

1. Grab the class and test name dynamically from the `request` object
2. Pass it into the Desired Capabilities object

```python
desired_caps["name"] = request.cls.__name__ + "." + request.function.__name__
```

Setting the Job Status

1. Use the `pytest_runtest_makereport` function to grab the test result
2. Append the result as an attribute to the `request` object
3. Use the result in the test teardown to pass the result to Sauce Labs

```python
# conftest.py
@pytest.hookimpl(tryfirst=True, hookwrapper=True)
def pytest_runtest_makereport(item, call):
    outcome = yield
    result = outcome.get_result()
    setattr(item, "result_" + result.when, result)

@pytest.fixture
def driver(request):
    # ...
    def quit():
        try:
            if config.host == "saucelabs":
                if request.node.result_call.failed:
                    driver_.execute_script("sauce:job-result=failed")
                    print "http://saucelabs.com/beta/tests/" + driver_.session_id
                elif request.node.result_call.passed:
                    driver_.execute_script("sauce:job-result=passed")
        finally:
            driver_.quit()
```

Chapter 3
Common Actions

Visit a page

```
driver.get("http://the-internet.herokuapp.com")
```

Find an element

Works using locators, which are covered in the next section.

```
# find just one, the first one Selenium finds
driver.find_element(locator)

# find all instances of the element on the page
driver.find_elements(locator)
# returns a collection
```

Work with a found element

```
# Chain actions together
driver.find_element(locator).click()

# Store the element
element = driver.find_element(locator)
element.click()
```

Perform an action

```
element.click()                   // clicks an element
element.submit()                  // submits a form
element.clear()                   // clears an input field of its text
element.send_keys("input text")  // types into an input field
```

Ask a question

Each of these returns a Boolean.

```
element.is_displayed()    // is it visible?
element.is_enabled()      // can it be selected?
element.is_selected()     // is it selected?
```

Retrieve information

```
# by attribute name
element.get_attribute("href")

# directly from an element
element.get_text()
```

For more info:

- the WebElement documentation

Chapter 4
Locators

Guiding principles

Good Locators are:

- unique
- descriptive
- unlikely to change

Be sure to:

1. Start with ID and Class
2. Use CSS selectors (or XPath) when you need to traverse
3. Talk with a developer on your team when the app is hard to automate
    1. tell them what you're trying to automate
    2. work with them to get more semantic markup added to the page

ID

```
driver.find_element(By.ID, "username")
```

Class

```
driver.find_element(By.CLASS_NAME, "dues")
```

CSS Selectors

```
driver.find_element(By.CSS_SELECTOR, "#example")
```

Approach	Locator	Description
ID	`#example`	`#` denotes an ID
Class	`.example`	`.` denotes a Class
Classes	`.flash.success`	use `.` in front of each class for multiple
Direct child	`div > a`	finds the element in the next child
Child/subschild	`div a`	finds the element in a child or child's child
Next sibling	`input.username + input`	finds the next adjacent element
Attribute values	`form input[name='username']`	a great alternative to id and class matches
Attribute values	`input[name='continue'][type='button']`	can chain multiple attribute filters together
Location	`li:nth-child(4)`	finds the 4th element only if it is an li
Location	`li:nth-of-type(4)`	finds the 4th li in a list
Location	`*:nth-child(4)`	finds the 4th element regardless of type
Sub-string	`a[id^='beginning_']`	finds a match that starts with (prefix)
Sub-string	`a[id$='_end']`	finds a match that ends with (suffix)
Sub-string	`a[id*='gooey_center']`	finds a match that contains (substring)
Inner text	`a:contains('Log Out')`	an alternative to substring matching

NOTE: Older browser (e.g., Internet Explorer 8) don't support CSS Pseudo-classes, so some of these locator approaches won't work (e.g., Location matches and Inner text matches).

For more info see one of the following resources:

- CSS Selector Game
- CSS & XPath Examples by Sauce Labs
- The difference between nth-child and nth-of-type
- CSS vs. XPath Selenium benchmarks
- CSS Selectors Reference
- XPath Syntax Reference

Chapter 5
Exception Handling

1. Try the action you want
2. Catch the relevant exception and return `false` instead

```
try:
    self._find(locator).is_displayed()
except NoSuchElementException:
    return False
return True
```

For more info see:

- a full list of Selenium exceptions

Chapter 6

Waiting

Implicit Wait

- Specify a timeout in seconds (typically during test setup)
- For every command that Selenium is unable to complete, it will retry it until either:
  - the action can be accomplished, or
  - the amount of time specified has been reached and raise an exception (typically `NoSuchElementException`)
- Less flexible than explicit waits
- Not recommended

```
driver.implicitly_wait(15)
```

Explicit Waits

- Specify a timeout (in seconds) and an expected condition to wait for
- Selenium will check for the expected condition repeatedly until either:
  - is is successful, or
  - the amount of time specified has been reached and raise an exception
- Recommended way to wait in your tests

```
wait = WebDriverWait(self.driver, timeout)
wait.until(
    expected_conditions.visibility_of_element_located(
      (locator['by'], locator['value'])))
```

For more info:

- Explicit vs Implicit Waits
- Selenium Python bindings documentation for explicit waits