AGENOR
delivering change, managing risk

Guide to Learning Selenium WebDriver with BDD

Table of Contents

Section 1: Introduction to Selenium

What is Selenium?

Selenium is an open-source framework widely used for automating web browsers. It provides a suite of tools and libraries that enable testers and developers to automate web-based applications across different browsers and platforms.

Selenium allows users to interact with web elements, simulate user actions, extract data, and perform various testing tasks. It supports multiple programming languages, including Python, Java, C#, and more, making it accessible to a wide range of developers and testers.

Selenium components: WebDriver, Grid, IDE

Selenium consists of several key components that contribute to its powerful automation capabilities. The first and most essential component is WebDriver. WebDriver provides a programming interface for interacting with web browsers, allowing users to automate browser actions such as clicking buttons, filling forms, navigating through pages, and extracting data. WebDriver supports various browsers, including Chrome, Firefox, Safari, and Edge, enabling cross-browser testing.

Another important component of Selenium is Selenium Grid. Selenium Grid allows users to distribute their tests across multiple machines and browsers simultaneously, making it ideal for running tests in parallel and achieving faster test execution. By leveraging Selenium Grid, users can scale their test suites, perform cross-platform testing, and reduce the overall test execution time.

Additionally, Selenium IDE (Integrated Development Environment) serves as a record-and-playback tool for creating and executing automated tests. It provides a simple and user-friendly interface that allows testers to record their interactions with a web application and generate corresponding test scripts. Selenium IDE is particularly useful for quick prototyping, generating test scripts for simple scenarios, and getting started with Selenium.

These components, WebDriver, Grid, and IDE, work together to provide a comprehensive suite for web browser automation and testing, catering to the diverse needs of developers and testers in their automation journeys.

In this guide we will be looking at WebDriver exclusively and in particular its use with Behaviour Driven Development (BDD).

Benefits of using Selenium WebDriver for automated testing

Selenium WebDriver offers several compelling benefits that make it a preferred choice for testers and developers. First and foremost, WebDriver provides excellent cross-browser compatibility, allowing tests to be executed on different browsers and ensures that applications are thoroughly tested across various browser environments, guaranteeing consistent behaviour and user experience.

Furthermore, WebDriver supports complex interactions and advanced features, including handling pop-ups, iframes, alerts, and multiple windows. It also provides support for handling JavaScript-driven web applications, making it suitable for testing modern, dynamic web pages. WebDriver's ability to handle complex scenarios and its rich set of features empower testers to create comprehensive and sophisticated test suites.

Additionally, Selenium WebDriver integrates seamlessly with various testing frameworks and tools, making it compatible with popular testing frameworks like TestNG and JUnit. This allows testers to leverage the capabilities of these frameworks to organize and execute tests efficiently, generate detailed reports, and perform test management.

Overall, Selenium WebDriver offers numerous benefits such as cross-browser compatibility, language support, robust API, advanced interactions, and integration with testing frameworks. These advantages make it a powerful tool for automating web application testing, increasing test coverage, improving efficiency, and ensuring the quality and reliability of web applications.

Some drawbacks and limitations of Selenium WebDriver

While Selenium is a powerful tool for test automation, it is important to be aware of its drawbacks. Here are some common drawbacks of using Selenium in test automation:

1. Limited Support for Desktop Applications and Mobile Apps: Selenium is primarily designed for web application testing and has limited support for automating desktop applications or mobile apps. If you need to automate testing for desktop or mobile platforms, you may need to use additional tools or frameworks alongside Selenium.

2. Time-Consuming Test Execution: Selenium tests can be time-consuming, especially when dealing with complex scenarios or large test suites. The time taken to interact with elements, navigate through pages, and wait for elements to load can add up, resulting in longer test execution times. This can be a challenge when you need to run tests frequently or as part of continuous integration.

3. Fragile Test Scripts: Selenium tests can be sensitive to changes in the application under test or the test environment. Even minor changes to the UI layout, element locators, or application behaviour can cause tests to fail, requiring updates to the test scripts. This fragility can result in maintenance overhead and make tests more prone to breakage.

4. Lack of Support for CAPTCHA and OTP: Selenium struggles to automate challenges like CAPTCHA or One-Time Password (OTP) verification, which are commonly used to prevent automated interactions. These security measures are intentionally designed to prevent automated scripts from accessing certain features, and automating tests involving CAPTCHA or OTP can be challenging with Selenium alone.

5. Steep Learning Curve: Selenium has a steep learning curve, especially for beginners or testers who are new to test automation. It requires knowledge of programming languages, web technologies, and understanding of the underlying concepts of WebDriver and locators. This learning curve can make it more time-consuming to get started with Selenium-based test automation.

6. Lack of Reporting and Test Management Features: Selenium itself does not provide built-in reporting or test management features. While there are third-party frameworks and tools available to enhance reporting capabilities, you may need to invest additional effort in setting up and configuring these tools to generate comprehensive test reports and manage test execution.

Despite these drawbacks, Selenium remains a popular and widely-used tool for web application test automation. It offers a robust set of features and a large community of users and contributors. By being aware of the limitations and leveraging additional tools and best practices, you can mitigate these drawbacks and maximize the benefits of using Selenium in your test automation efforts.

Further Reading:

Selenium official website:
https://www.selenium.dev

Selenium WebDriver documentation:
https://www.selenium.dev/documentation/webdriver/

Section 2: Learning Selenium

Launching a Browser and Navigating to a URL

Since Selenium 4.6 our lives have been made easier as Selenium now takes care of driver management and we no longer have to download or maintain our own drivers.

Assuming you have downloaded Selenium and imported the WebDriver package (detailed in a later section) then you can launch a browser, for example Chrome, and navigate to a URL with code similar to this,

```
driver = webdriver.Chrome()
driver.get(“https://www.google.com”)
```

Remember, once you have finished your test you will need to use **‘driver.quit()’** to close your browser session.

Learn more and get an overview on the sections to follow here:
https://www.selenium.dev/documentation/webdriver/getting_started/first_script/

Locating elements: By class, ID, name, CSS selector, XPath

Locating elements is a fundamental aspect of web automation testing with Selenium WebDriver. It involves identifying and interacting with specific elements on a web page, such as buttons, input fields, dropdowns, and links. WebDriver provides various methods and strategies for element location, giving testers the flexibility to choose the most appropriate approach based on the characteristics of the application under test.

The most common method for element location is using locators based on HTML attributes such as ID, name, class, CSS selectors, and XPath. These locators provide unique identifiers for elements, making it easier for WebDriver to find and interact with them. For example, an ID locator targets an element using its unique ID attribute, while a CSS selector locates elements based on their CSS properties and attributes. XPath locators, on the other hand, use a path expression to navigate through the XML structure of the HTML document to identify elements.

WebDriver also supports more advanced locating techniques, such as locating elements by their text content, link text, partial link text, and tag name. These methods allow testers to locate elements based on their visible text or the text contained within specific HTML tags. They are particularly useful when elements do not have distinct attributes but can be identified based on their textual content.

In addition to these built-in locating strategies, WebDriver allows testers to create custom locators using JavaScript and execute scripts to locate elements dynamically. This approach is useful when elements have dynamic IDs or when the structure of the web page changes dynamically based on user interactions. By executing custom scripts, testers can locate elements based on complex conditions, calculations, or dynamic patterns.

In summary, locating elements is a critical aspect of Selenium WebDriver automation. It involves using various locating strategies such as HTML attributes, CSS selectors, XPath, text content, and tag names. Testers can also create custom locators using JavaScript to handle dynamic elements. Understanding and utilizing these techniques effectively empowers testers to accurately identify and interact with elements during automated testing, ensuring the reliability and robustness of web applications.

Learn about locator strategies here:
https://www.selenium.dev/documentation/webdriver/elements/locators/

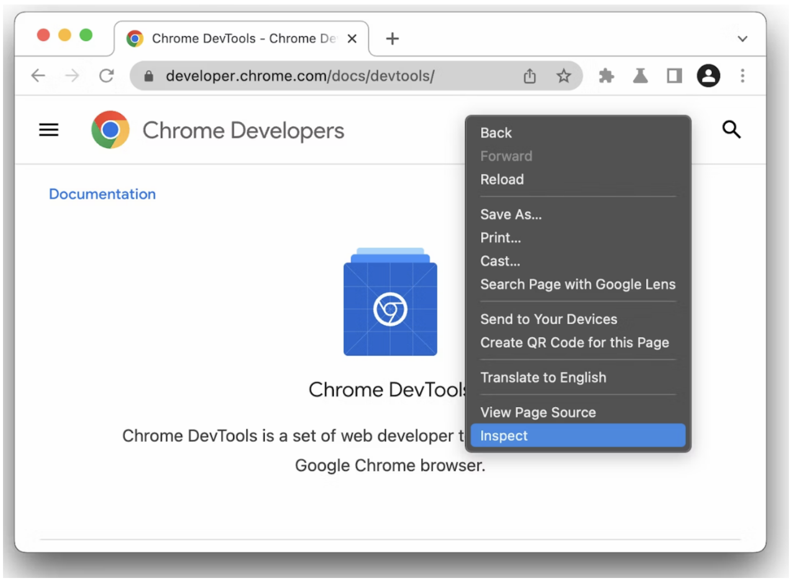
and finding web elements here:
https://www.selenium.dev/documentation/webdriver/elements/finders/

Using Chrome DevTools to locate web elements

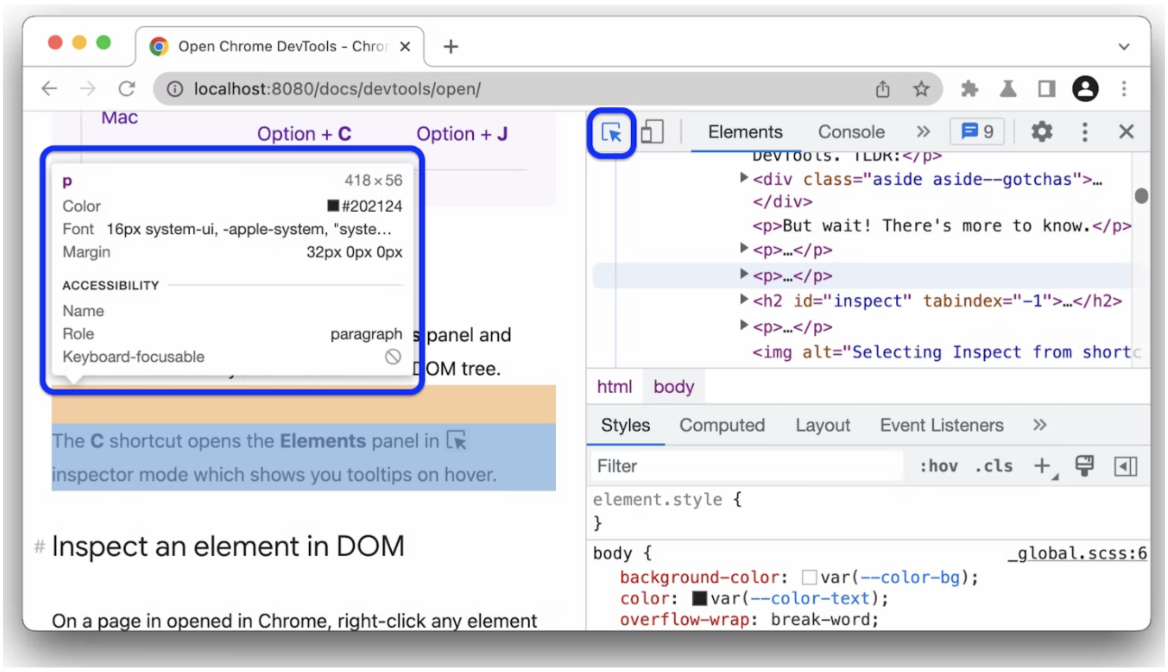
One powerful feature of Chrome DevTools is its ability to help testers locate web elements on a page. This can be especially useful when identifying unique attributes or properties that can be used in automated tests. Let's explore the steps to use Chrome DevTools for element inspection and identification.

Step 1: Open Chrome DevTools
To begin, open the Chrome browser and navigate to the webpage you want to inspect. Right-click anywhere on the page and select "Inspect" from the context menu (see next page).

Alternatively, you can press `**Ctrl + Shift + I**` (Windows/Linux) or `**Cmd + Option + I**` (Mac) to open Chrome DevTools or just `**F12**`.



Step 2: Select the Element Inspection Tool
Once Chrome DevTools is open, you'll see a panel with various tabs. Click on the icon resembling a cursor in a square, known as the "Element Inspection Tool." This tool allows you to hover over elements on the page and inspect their properties.


Open Chrome DevTools - Chro
localhost:8080/docs/devtools/open/
Mac
Option + C
Option + J
p 418 × 56
Color #202124
Font 16px system-ui, -apple-system, "syste...
Margin 32px 0px 0px
ACCESSIBILITY
Name
Role paragraph
Keyboard-focusable
panel and
OM tree.
The C shortcut opens the Elements panel in inspector mode which shows you tooltips on hover.
Inspect an element in DOM
On a page in opened in Chrome, right-click any element
Elements
Console
9
html
body
Styles
Computed
Layout
Event Listeners
Filter
:hov
.cls

```
<div class="aside aside--gotchas">…
</div>
<p>But wait! There's more to know.</p>
<p>…</p>
<p>…</p>
<h2 id="inspect" tabindex="-1">…</h2>
<p>…</p>
<img alt="Selecting Inspect from short
```

```
element.style {
}
body {                                    _global.scss:6
   background-color: var(--color-bg);
   color: var(--color-text);
   overflow-wrap: break-word;
```

Step 3: Locate and Inspect Elements
With the Element Inspection Tool active, hover over different elements on the page. As you hover over an element, it will be highlighted in the browser window, and its corresponding HTML code will be displayed in the Elements tab of Chrome DevTools. You can also right-click on an element in the Elements tab and choose "Inspect" to focus on it.

Step 4: Identify Unique Attributes or Properties
Within the Elements tab of Chrome DevTools, you can explore the HTML code to identify unique attributes or properties of the element you're interested in. Look for attributes like `id`, `class`, `name`, or any custom attribute that uniquely identifies the element. These attributes can be used to locate the element in your automated tests.

Step 5: Copy Element Selector
Chrome DevTools also provides the option to copy the element selector, which generates a unique CSS selector for the selected element. Right-click on the highlighted element in the Elements tab, hover over "Copy", and select "Copy selector." This selector can be pasted into your test code for element identification.

Using Chrome DevTools for element inspection and identification provides a visual and interactive way to locate web elements. By understanding the HTML structure and identifying unique attributes or properties, testers can effectively leverage this tool to locate elements for their automated tests.

Interacting with elements: Clicking, typing, selecting options, etc.

Interacting with elements is a fundamental part of web automation testing using Selenium WebDriver. Once elements are located, testers can perform various actions such as clicking, typing, selecting options, and more. These actions simulate user interactions with the web application and enable comprehensive testing of its functionality and user experience.

Clicking on elements is a common action performed during test scenarios. It involves simulating a mouse click on buttons, links, checkboxes, radio buttons, and other clickable elements. WebDriver provides the **`click()`** method to perform this action on elements. By clicking on elements, testers can trigger events, navigate to different pages, submit forms, or verify the expected behaviour of interactive elements.

Typing or entering text into input fields is another crucial action for data entry and form submission. WebDriver provides the **`send_keys()`** method to simulate keyboard input. Testers can use this method to enter text, numbers, or special characters into input fields. They can also combine it with other actions like clearing existing text, **`clear()`**, before typing new values. This interaction ensures that the application correctly handles user input and form submissions.

Handling different element types: Buttons, checkboxes, dropdowns etc.

Selecting options from dropdown lists or dropdown menus is a common task in web applications. WebDriver offers dedicated methods to handle dropdowns, such as **`select_by_visible_text()`**, **`select_by_value()`**, and **`select_by_index()`**. These methods allow testers to choose specific options based on their visible text, attribute values, or index position. By selecting options, testers can verify the functionality and behaviour of dropdowns, cascading menus, and multi-select dropdowns.

Interacting with checkboxes and radio buttons is crucial for testing forms, preferences, and options. WebDriver provides the **`click()`** method to toggle the state of checkboxes and radio buttons. By checking or unchecking checkboxes and selecting radio buttons, testers can validate the behaviour of these input elements, including default selections, exclusivity among radio buttons, and multiple selections with checkboxes.

Learn about interacting with elements here:
https://www.selenium.dev/documentation/webdriver/elements/interactions/

Handling alerts and pop-ups

Handling alerts, pop-ups, and modals is also an important aspect of interacting with elements. WebDriver offers methods like **`switch_to.alert`** and **`switch_to.window`** to handle these elements. Testers can accept or dismiss alerts, switch between windows, and interact with the content within modals. This capability allows comprehensive testing of applications that utilize various types of pop-ups and modals for notifications, confirmations, and user interactions.

Learn about handling alerts here:
https://www.selenium.dev/documentation/webdriver/interactions/alerts/

Working with frames and windows

Frames, also known as iframes, are HTML elements that allow embedding one HTML document within another. Windows refer to browser windows or tabs that can contain separate web pages. To interact with elements inside frames or windows, testers need to switch the WebDriver's focus to the desired frame or window.

Switching focus to a frame involves using the **`switch_to.frame()`** method provided by WebDriver. Testers can switch to a frame by specifying the frame element, index, or name/id attribute. Once inside the frame, WebDriver commands will target elements within that frame. It's important to note that when working with nested frames, testers may need to switch the focus to the parent frame or the default content before switching to the desired frame.

WebDriver also provides methods to switch focus between windows. The **`window_handles`** attribute returns a list of window handles, each representing a unique window or tab. Testers can use the **`switch_to.window()`** method by passing the desired window handle to switch the focus to that window. By switching between windows, testers can interact with elements, perform verifications, and validate the behaviour of the application across multiple windows or tabs.

When working with frames or windows, it's important to note that WebDriver maintains a context stack to keep track of the current frame or window. Testers need to switch the context appropriately to perform actions on the desired elements. Additionally, when switching focus between frames or windows, testers should handle any synchronization issues, allowing sufficient time for the new content to load before interacting with elements.

Learn about working with frames here:
https://www.selenium.dev/documentation/webdriver/interactions/frames/

and windows here:
https://www.selenium.dev/documentation/webdriver/interactions/windows/

Waiting strategies: Implicit wait, explicit wait, fluent wait

Waiting is an essential aspect of automated testing, especially when dealing with dynamic web applications where elements may not be immediately available. Selenium provides several waiting strategies to ensure that test scripts wait for the required elements to appear before proceeding with further actions. Let's explore three common waiting strategies: implicit wait, explicit wait, and fluent wait.

1. Implicit Wait:
The implicit wait is a global waiting strategy applied to the entire lifespan of the WebDriver instance. When set, it tells the WebDriver to wait for a certain amount of time before throwing a NoSuchElementException if an element is not immediately found. This wait is applied for every element lookup across the test script. The implicit wait is set using the **`driver.implicitly_wait(timeout)`** method, where **`timeout`** represents the maximum amount of time to wait for an element to be present. It is important to note that the implicit wait is set once and affects all subsequent element searches. Best practice is not to use implicit waits if possible.

2. Explicit Wait:
The explicit wait provides more fine-grained control over waiting for specific conditions to be met before proceeding with test execution. Unlike the implicit wait, the explicit wait is applied to specific elements or conditions only when needed. It allows test scripts to wait until a certain condition is satisfied before performing the next action. This condition can be based on various factors such as element visibility, presence, clickability, etc. The WebDriverWait class in Selenium is used for implementing explicit waits. It takes the WebDriver instance and the maximum time to wait as parameters. You can then specify the expected condition using the **`ExpectedConditions`** class. For example, **`WebDriverWait(driver, timeout).until(EC.visibility_of_element_located((By.ID, 'element-id')))`**.

3. Fluent Wait:
The fluent wait is another flexible waiting strategy in Selenium that allows test scripts to wait for an element based on specific conditions. It provides more control over the polling interval, maximum wait time, and exceptions to ignore during the wait. The fluent wait allows you to define custom polling conditions and timeouts. It is useful when dealing with complex scenarios where elements may appear or disappear dynamically. The FluentWait class is used to implement fluent waits in Selenium. It provides methods to set the timeout, polling interval, and the expected conditions. For example,

`FluentWait(driver).withTimeout(timeout, TimeUnit.SECONDS).pollingEvery(interval, TimeUnit.MILLISECONDS).until(condition)`.

By utilizing these waiting strategies effectively, testers can ensure that their automated tests are resilient to dynamic web application behaviour. Each waiting strategy has its own benefits and can be applied based on the specific requirements of the test scenario.

Learn about waits here:
https://www.selenium.dev/documentation/webdriver/waits/

Section 3: Selenium Practice

Setting up your coding environment

The framework we will be working with is written in Python so we will need to install this onto our machines and also download a code editor to allow us to start working with it.

There are two popular choices, Pycharm (https://www.jetbrains.com/pycharm/download/) which is a Python specific code editor and VS Code which offers support for multiple languages.

I recommend VS Code as a good option to get experience with as it's not guaranteed you will only be using Python in the future. The following exercises will all use VS Code in the examples given.

Installing Python

Go to https://www.python.org/downloads and download the version you need for the operating system you are on.

Follow the installer instructions.

Windows

- Note the path where Python is being installed - generally C:\Users(Your logged in User)\AppData\Local\Programs\Python\PythonXX

Set Python home in Environment Variables

- Do a search for 'Edit the system environment variables' in Windows
- Click on Environment Variables
- Edit 'Path'
- Set a new path to match the path where Python was installed
- Repeat above and add path for, e.g. C:\Users(Your logged in User)\AppData\Local\Programs\Python\PythonXX/Scripts
- Open a new Command Prompt and enter: python --version
- If a response is returned then everything is installed fine

Mac

Path variable is set automatically by MacOS on install

- Open a new terminal and enter: which python3
- If a response is returned with a path then everything is installed OK

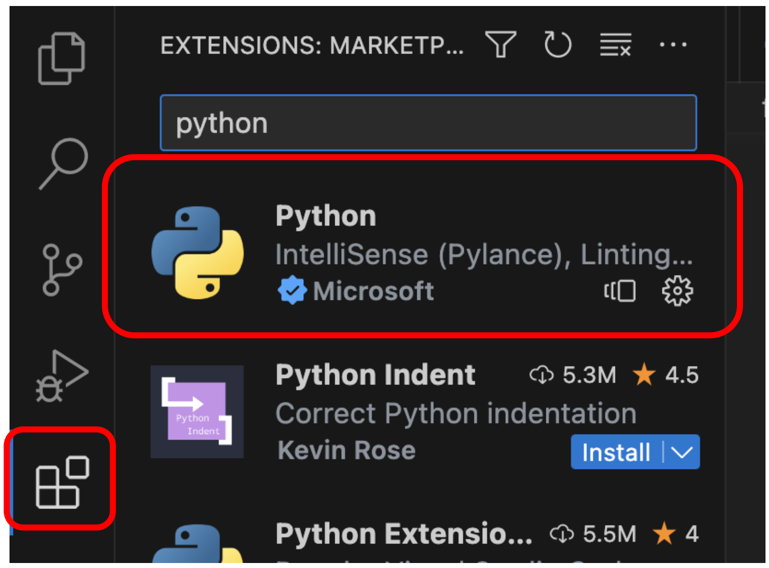
Installing VS Code

https://code.visualstudio.com/download

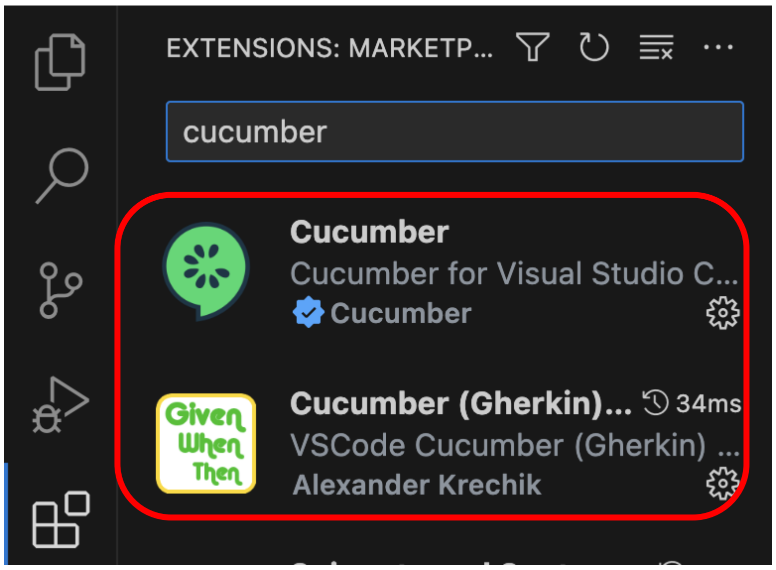
Go to the above link and download the version you need for the operating system you are on.

Now we need to download some extensions so we can work with Python and also some other extensions for later when we start using BDD tools.

Open up VS Code, in the left-hand margin click on the Extensions icon and then search for Python. Click on the 'Install' button for Python by Microsoft:



Next, do a search for cucumber and install the Cucumber and Cucumber (Gherkin) extensions:



That's all we really need to get up and running, however, there are many more extensions you can explore. Below are a couple more I recommend so that VS Code on your machine looks similar to the examples given later:

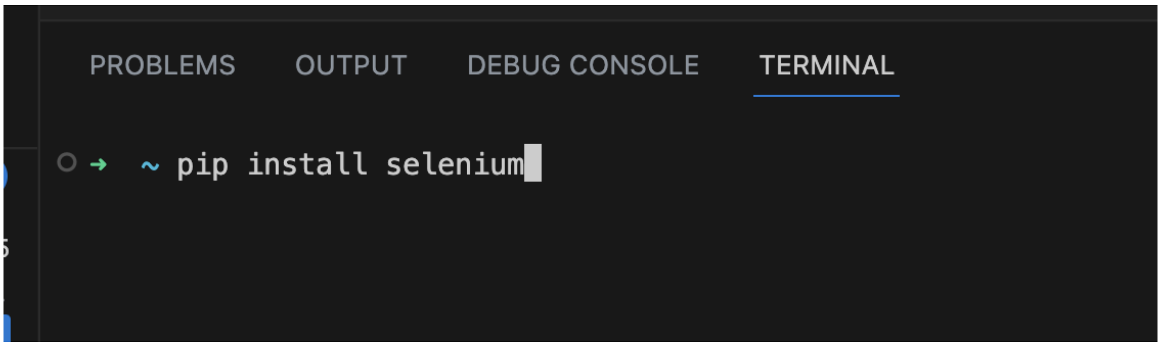
1. vs-code icons: makes all your folders and files look cool
2. Prettier: a popular code auto formatter

Installing and using Selenium

Now that we have Python and our code editor installed let's take a look at some Selenium basics. First, we need to install Selenium for Python.

In VS Code, from the menu bar open up a new terminal (Terminal > New Terminal).

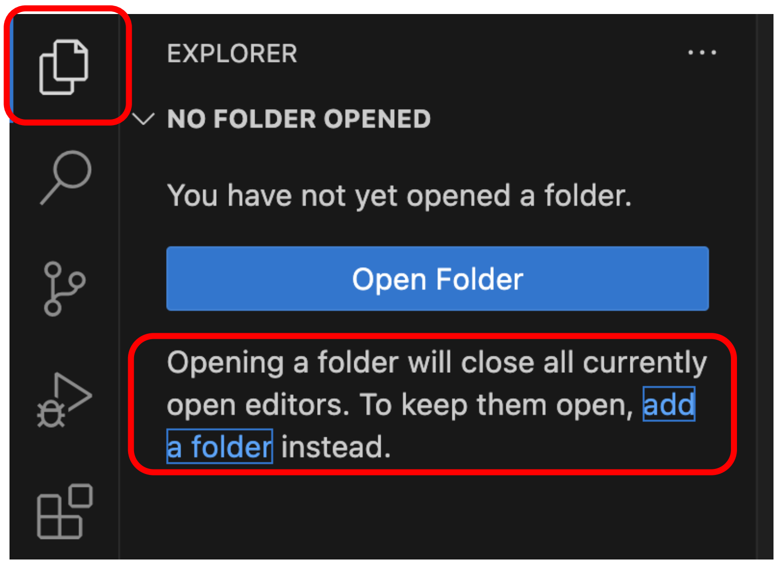
For Windows, enter `pip install selenium`. For Mac, enter `pip3 install selenium`. You should see all the required dependencies and packages being installed.



```
○ ➜  ~ pip install selenium
```

Your first script

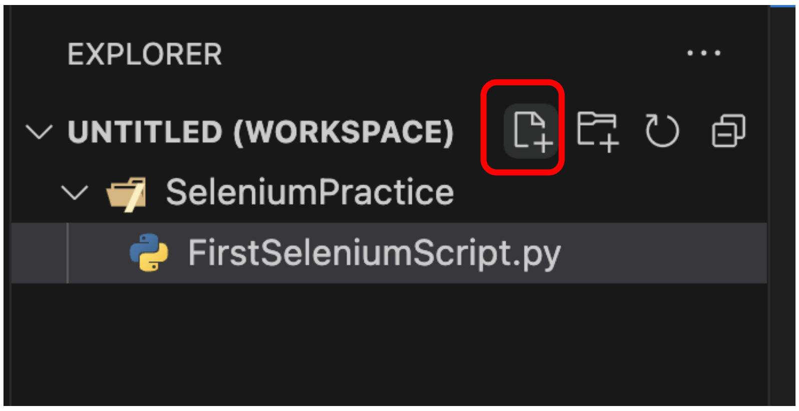
From the left-hand menu select the Explorer icon and then click on 'add a folder`:



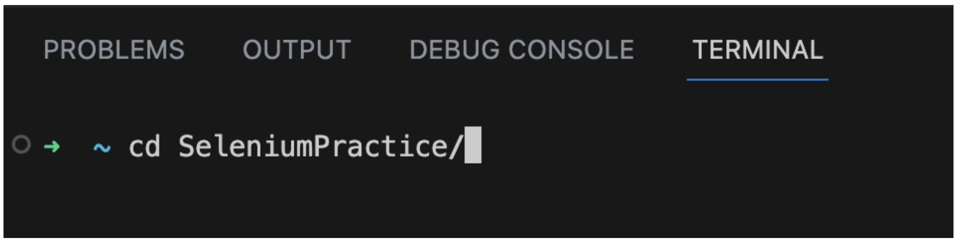
You have not yet opened a folder.

Opening a folder will close all currently open editors. To keep them open, add a folder instead.

Create a new folder called `SeleniumPractice`. In VS Code, highlight your new folder and click on the new file icon. Create a new python file called `FirstSeleniumScript.py`:

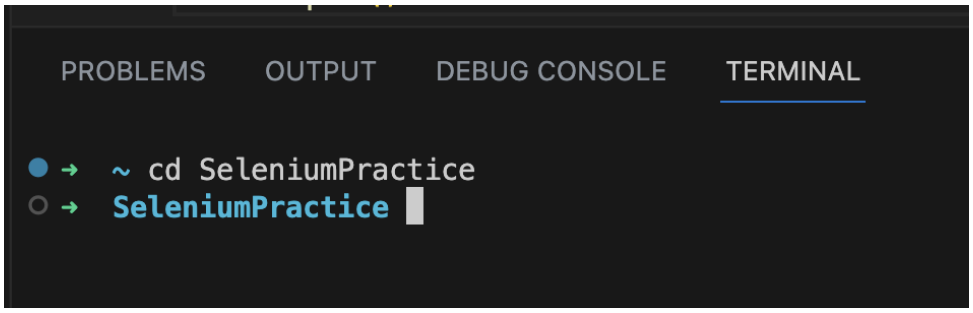


We now need to change our working directory so we are in our SeleniumPractice directory (or package as it's known in Python). In your code editor terminal enter `cd SeleniumPractice` (or the full path to your directory depending where you saved it), e.g.:



```
○ ➜  ~ cd SeleniumPractice/
```

You should now be in the correct working directory:



```
● ➜  ~ cd SeleniumPractice
○ ➜  SeleniumPractice
```

Import Selenium Packages

The following steps are based on the official guide at
https://www.selenium.dev/documentation/webdriver/getting_started/first_script/

Open your `FirstSeleniumScript.py` file (or module in Python) and type in the following:

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

These imports are necessary so we can use WebDriver and element locators in our script.

Initialize driver and navigate to URL

Next, we need initialize a driver so we can work with a web browser, in this example Chrome:

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

We can now use this driver to navigate to a URL of our choice:

```
driver.get("https://www.selenium.dev/selenium/web/web-form.html")
```

More details on browser navigation can be found here:
https://www.selenium.dev/documentation/webdriver/interactions/navigation/

Browser waits

Synchronizing our code with the current state of the browser is one of the biggest challenges with Selenium, and doing it well is an advanced topic.

Essentially you want to make sure that the element is on the page before you attempt to locate it and the element is in an interactable state before you attempt to interact with it.

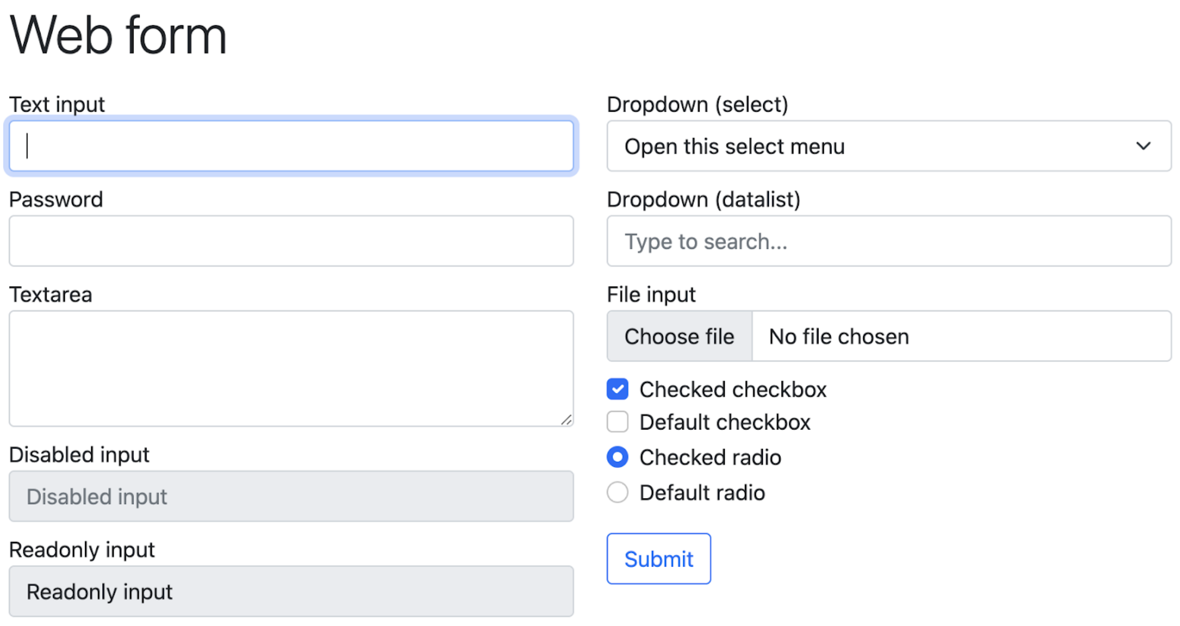
An implicit wait is rarely the best solution, but it’s the easiest to demonstrate here, so we’ll use it as a placeholder:

```
driver.implicitly_wait(1)
```

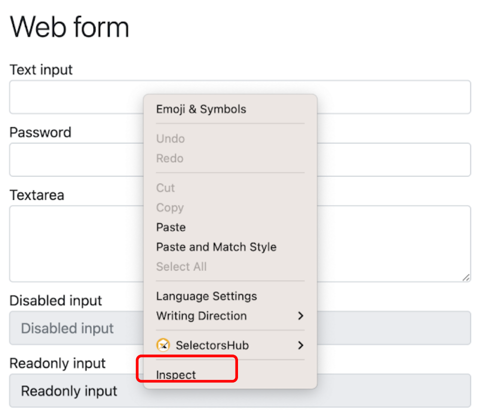
Further documentation on browser waits can be found here:
https://www.selenium.dev/documentation/webdriver/waits/

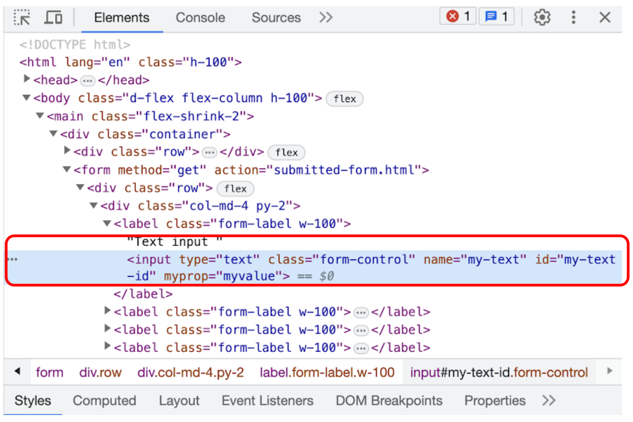
Locating web elements

Let’s take a look at our web page (https://www.selenium.dev/selenium/web/web-form.html) and see how we might want to interact with it. For instance, we may want to select and enter text into the `Text input` field and then click `Submit`:



Right click on the `Text input` field to bring up the context menu and select `Inspect`. This will open up Chrome DevTools and highlight the HTML attributes of the element we selected:





```
<!DOCTYPE html>
<html lang="en" class="h-100">
 <head>…</head>
 <body class="d-flex flex-column h-100"> flex
  <main class="flex-shrink-2">
   <div class="container">
    <div class="row">…</div> flex
    <form method="get" action="submitted-form.html">
     <div class="row"> flex
      <div class="col-md-4 py-2">
       <label class="form-label w-100">
         "Text input "
         <input type="text" class="form-control" name="my-text" id="my-text
         -id" myprop="myvalue"> == $0
       </label>
       <label class="form-label w-100">…</label>
       <label class="form-label w-100">…</label>
       <label class="form-label w-100">…</label>
```

form div.row div.col-md-4.py-2 label.form-label.w-100 input#my-text-id.form-control

Taking a closer look at the element’s attributes we can see the following:

```
<input type="text" class="form-control" name="my-text" id="my-text
-id" myprop="myvalue"> == $0
```

Selenium offers us several methods to locate an element. It's a good idea to get very familiar with these because they are a vital component of using Selenium.

For a full list consult the documentation here:
https://www.selenium.dev/documentation/webdriver/elements/locators/

Going back to our example, we are looking for an HTML attribute (or locator) that will uniquely identify our element. Since 'id' is supposed to be unique per page it seems like this will be an ideal candidate, i.e., the id="my-text-id" attribute.

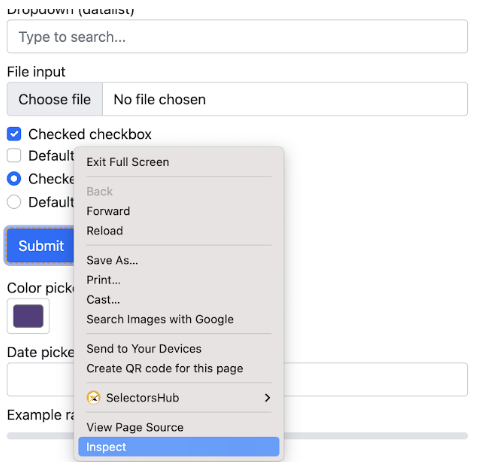
The syntax in Python for telling the driver to locate an element by 'id' and assigning it to a variable to use later on is:

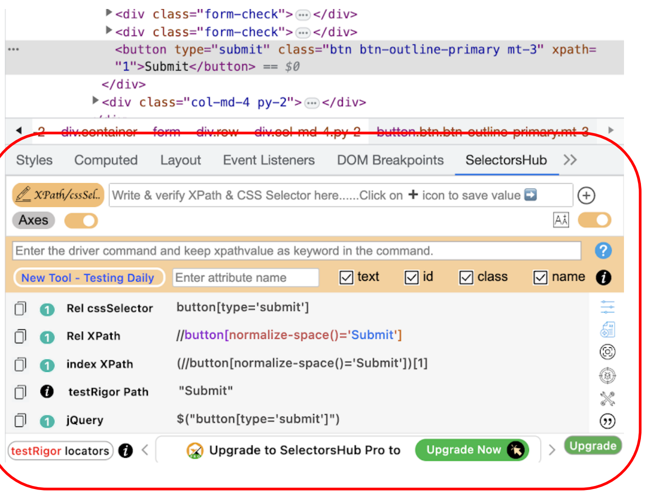
```python
text_box = driver.find_element(By.ID, "my-text-id")
```

Now I want to tell you about a handy tool that helps to make identifying unique locators on a page easy.

It's a Chrome extension called SelectorsHub and can be installed here:
https://selectorshub.com/selectorshub/

If we find our `Submit' button on the web page and right click to inspect it, if we have SelectorsHub installed we'll see the following:




```
▸<div class="form-check">…</div>
▸<div class="form-check">…</div>
 <button type="submit" class="btn btn-outline-primary mt-3" xpath=
 "1">Submit</button> == $0
</div>
▸<div class="col-md-4 py-2">…</div>
```
div.container form div.row div.col-md-4.py-2 button.btn.btn-outline-primary.mt-3
Styles
Computed
Layout
Event Listeners
DOM Breakpoints
SelectorsHub
XPath/cssSel..
Write & verify XPath & CSS Selector here......Click on + icon to save value
Axes
Enter the driver command and keep xpathvalue as keyword in the command.
New Tool - Testing Daily
Enter attribute name
text
id
class
name

1	Rel cssSelector	button[type='submit']
1	Rel XPath	//button[normalize-space()='Submit']
1	index XPath	(//button[normalize-space()='Submit'])[1]
i	testRigor Path	"Submit"
1	jQuery	$("button[type='submit']")

testRigor locators
Upgrade to SelectorsHub Pro to
Upgrade Now
Upgrade

SelectorsHub will give us a list of locators and tell us if it is unique on the page, indicated by the green circled '1'. We can scroll down and choose which locator we'd prefer to use (note, not all are applicable to Selenium).

In our example, the Rel cssSelector is one available for us to use in Selenium and looks like a good candidate.

The syntax in Python for telling the driver to locate an element by its CSS selector and assigning it to a variable to use later on is:

```
submit_button = driver.find_element(By.CSS_SELECTOR, "button[type='submit']")
```

Interacting with web elements

There are only a few actions we can take on an element but they are used frequently and again you should become very familiar with them.

For a full list consult the documentation here:
https://www.selenium.dev/documentation/webdriver/elements/interactions/

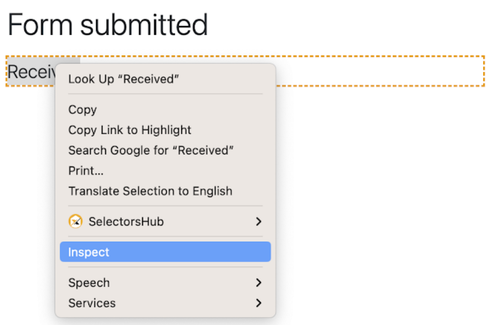
Going back to our example, we want to enter some text into the text box and then click the submit button. The syntax in Python for this (using our variables from the previous step) is:

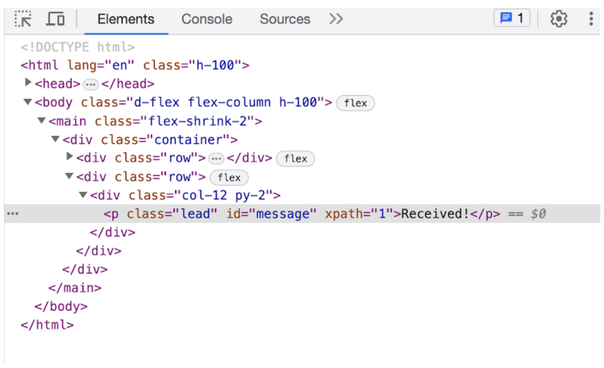
```
text_box.send_keys("Selenium")
submit_button.click()
```

Asserting if our test has passed

After clicking the submit button, we are navigated to a page confirming our form has been submitted. Let's say the point of our test was to confirm after we submit a form then we receive a message saying 'Received!'.

Let's right click on the 'Received!' text and see if we can find a suitable locator:





```
<!DOCTYPE html>
<html lang="en" class="h-100">
 ▶<head>…</head>
 ▼<body class="d-flex flex-column h-100"> flex
   ▼<main class="flex-shrink-2">
     ▼<div class="container">
       ▶<div class="row">…</div> flex
       ▼<div class="row"> flex
         ▼<div class="col-12 py-2">
             <p class="lead" id="message" xpath="1">Received!</p> == $0
           </div>
         </div>
       </div>
     </main>
   </body>
</html>
```

Again, we see the element has id="message". In general, we should favour using IDs where we can as they should be unique to a page and less subject to change than using Xpath or CSS Selectors.

Selenium also allows us to access information stored in an element. For instance, we can grab the text from within the element and use it to assert an expected result. The syntax in Python to achieve all this is:

```
message = driver.find_element(By.ID, "message")
value = message.text
assert value == "Received!"
```

To learn more about other Selenium methods that allow us to retrieve information can be found here:
https://www.selenium.dev/documentation/webdriver/elements/information/

Ending the session

Once our test is completed, we need to end the session and close our browser. The code for this is:

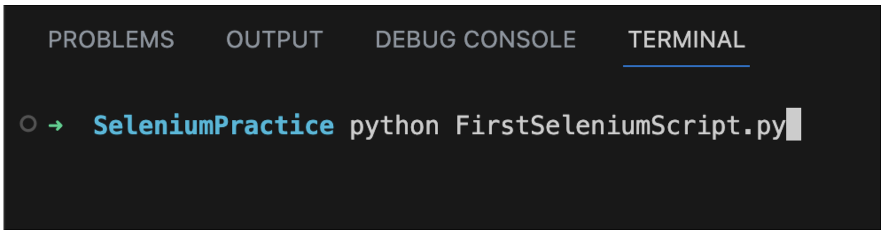
```
driver.quit()
```

Running the code

Our finished code should now look like this:

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


driver = webdriver.Chrome()

driver.get("https://www.selenium.dev/selenium/web/web-form.html")

driver.implicitly_wait(1)

text_box = driver.find_element(By.ID, "my-text-id")
submit_button = driver.find_element(By.CSS_SELECTOR, "button[type='submit']")

text_box.send_keys("Selenium")
submit_button.click()

message = driver.find_element(By.ID, "message")
value = message.text
assert value == "Received!"

driver.quit()
```

To run the test and check our code works, enter the below code into the terminal:



```
○ → SeleniumPractice python FirstSeleniumScript.py
```

Note, if you're using a Mac the command will be `python3 FirstSeleniumScript.py`.

You should see Selenium launch a Chrome browser, enter text and click the `Submit` button and then close down but blink and you'll miss it. Selenium operates pretty quickly!

Next Steps

Spend some time reading the documentation and experiment locating and interacting with different elements at: https://www.selenium.dev/selenium/web/web-form.html

Section 4: Introduction to BDD

What is BDD (Behaviour-Driven Development)?

Behaviour-Driven Development (BDD) is a software development approach that aims to bridge the gap between technical teams (developers, testers) and non-technical stakeholders (business analysts, product owners) by focusing on the behaviour of a system or application from the user's perspective.

In BDD, the emphasis is placed on describing the desired behaviour of the system using natural language that is understandable to both technical and non-technical individuals. This natural language specification is typically written in the form of executable scenarios called "feature files" or "specifications."

BDD promotes collaboration and shared understanding among team members by encouraging conversations around examples and scenarios that define the expected behaviour of the system. These scenarios are typically written using a domain-specific language (DSL) such as Gherkin, which provides a structured format for expressing the behaviour.

One of the key principles of BDD is the "Three Amigos" collaboration, which involves the participation of the business analyst/product owner, developer, and tester in discussing and refining the behaviour of the system. This collaboration ensures that everyone has a shared understanding of the requirements and that the system is built to meet the desired behaviour.

BDD also promotes the concept of "living documentation" where the executable scenarios serve as a single source of truth for the system's behaviour. These scenarios can be automated using testing frameworks and executed as part of the continuous integration and delivery pipeline, providing quick feedback on whether the system is behaving as expected.

By adopting BDD practices, teams can improve communication, reduce ambiguity in requirements, and focus on delivering value to the end-users by aligning development efforts with the desired behaviour of the system.

For further reading and resources on BDD, you can refer to the following online sources:
1. Cucumber Documentation: https://cucumber.io/docs/
2. Behave Documentation: https://behave.readthedocs.io/
3. The Cucumber Book: Behaviour-Driven Development for Testers and Developers: https://pragprog.com/titles/hwcuc/the-cucumber-book/

These resources provide detailed information and practical examples to help you understand and implement BDD in the software development process.

Key concepts: feature files, scenarios, step definitions

1. Feature Files:
Feature files are an essential component of BDD and serve as a communication and collaboration tool between the business stakeholders, developers, and testers. A feature file represents a specific feature or functionality of the system being developed. It is typically written in a plain-text format and uses a language called Gherkin.

Gherkin is a domain-specific language (DSL) that provides a structured way to define the behaviour of the system using keywords such as "Feature," "Scenario," "Given," "When," "Then," and others. Feature files are usually named with the `.feature` extension and contain a collection of scenarios that describe different aspects of the feature.

2. Scenarios:
Scenarios represent specific test cases or examples that describe the expected behaviour of the system. Each scenario consists of a series of steps written using Gherkin keywords. These steps outline the preconditions, actions, and expected outcomes of the test case.

Scenarios are typically independent and self-contained, focusing on a specific aspect of the feature. They are designed to be readable and understandable by both technical and non-technical stakeholders. Scenarios help to document and specify the behaviour of the system in a concise and structured manner.

3. Step Definitions:
Step definitions are the code implementation of the steps described in the feature files. They define the actions that need to be performed to execute the scenario and validate its expected outcome. Step definitions are written in a programming language such as Python, Java, or Ruby, depending on the chosen BDD framework.

Each step in a scenario is mapped to a corresponding step definition, which contains the logic to interact with the system under test, perform assertions, and handle the test execution flow. Step definitions facilitate the automation of the scenarios by providing the necessary instructions to execute the tests and validate the system's behaviour.

The step definitions interact with the underlying automation framework (e.g., Selenium WebDriver) to perform actions such as clicking buttons, entering text, selecting options, verifying element states, and more. They encapsulate the test logic and allow for reusability and maintainability of the test code.

By separating the feature files (specifications) from the step definitions (implementations), BDD promotes a clear distinction between the business requirements and the technical implementation, enabling collaboration between stakeholders with different skill sets.

Benefits of using BDD for test automation

Using BDD for test automation offers several benefits, including:

1. Enhanced Collaboration: BDD encourages collaboration between business stakeholders, developers, and testers. By using a common language and structured format (Gherkin), all team members can actively participate in defining and understanding the system's behaviour. This collaborative approach fosters better communication, reduces misunderstandings, and ensures that the developed software meets the desired business outcomes.

2. Improved Test Documentation: BDD promotes the creation of living documentation in the form of feature files. These feature files serve as executable specifications and provide a comprehensive understanding of the system's behaviour. They act as a single source of truth for the expected behaviour and serve as documentation that remains up-to-date throughout the development lifecycle. This documentation aspect facilitates knowledge sharing, onboarding of new team members, and overall maintainability of the test suite.

3. Clearer and Readable Test Scenarios: BDD encourages the use of a structured language (Gherkin) to define test scenarios. This language follows a natural language format that is easy to understand by both technical and non-technical stakeholders. The use of descriptive keywords such as Given, When, Then, and And helps to create self-explanatory scenarios, making it easier to review and validate the expected behaviour of the system.

4. Test Reusability and Maintainability: With BDD, test scenarios and step definitions can be designed to be modular and reusable. By breaking down the system's behaviour into small, independent scenarios, it becomes easier to reuse the steps across different scenarios and feature files. This reusability reduces duplication, enhances test maintenance, and promotes a more efficient test automation framework.

5. Focus on Business Value: BDD aligns the test automation efforts with the desired business outcomes. By involving business stakeholders in the creation and review of feature files, the testing process becomes more business-driven. Test scenarios are written from a user's perspective, focusing on the value delivered to the end user. This approach ensures that the developed software meets the business requirements and helps to identify potential gaps or misunderstandings early in the development cycle.

6. Early Bug Detection: BDD promotes early bug detection by encouraging collaboration and defining the expected behaviour upfront. By involving stakeholders in defining scenarios, any discrepancies or misunderstandings can be identified and resolved early in the process. This proactive approach to testing helps in catching issues at an early stage, reducing the overall cost and effort of fixing bugs in later stages of development.

7. Integration with Agile and Continuous Delivery: BDD integrates well with agile development methodologies and continuous delivery practices. The iterative and incremental nature of BDD allows for the continuous refinement of scenarios and step definitions as the software evolves. BDD aligns with the principles of frequent feedback,

collaboration, and fast feedback loops, making it a suitable approach for teams practicing agile and DevOps.

By leveraging the benefits of BDD, test automation teams can create a more collaborative, efficient, and business-focused approach to automated testing, resulting in higher quality software products and improved team productivity.

Section 5: BDD Test Frameworks

BDD test frameworks

Because of the benefits of BDD, many test frameworks at companies using agile methodologies will incorporate Selenium with BDD and typically include the following attributes:

1. Test Runner: The framework utilizes a test runner that executes the tests written in the BDD format. The test runner is responsible for parsing feature files, mapping them to corresponding step definitions, and executing the tests.

2. Feature Files: Feature files are written in a human-readable format and describe the behaviour or functionality being tested. They contain scenarios and associated steps written in a Given-When-Then format. Feature files serve as a communication tool between stakeholders and provide a high-level understanding of the system's behaviour.

3. Step Definitions: Step definitions define the actual implementation of the steps mentioned in the feature files. Each step in the feature file is mapped to a corresponding step definition, which interacts with the web elements using Selenium WebDriver methods. Step definitions encapsulate the logic and actions required to perform the test steps.

4. Page Objects: The framework incorporates the Page Object Model (POM) pattern to represent web pages as classes and encapsulate the interaction with web elements on those pages. Each web page has a corresponding page object class that contains methods to interact with the elements. Page objects provide a modular and reusable approach to managing web elements across different tests.

5. Test Data Management: The framework includes mechanisms to manage test data efficiently. This can be achieved through data-driven testing techniques, such as reading test data from external files (e.g., Excel, CSV) or databases. Test data is passed to the step definitions or page objects to drive test scenarios with different data sets.

6. Reporting: The framework generates comprehensive test reports that provide insights into the test execution results. These reports include information about test scenarios, pass/fail status, execution time, and any error or exception messages. Reporting helps in tracking test progress, identifying issues, and facilitating collaboration among team members.

7. Configuration Management: The framework allows for the configuration of various settings, such as the choice of browsers, timeouts, logging levels, and other environment-specific parameters. Configuration management ensures flexibility and adaptability of the framework across different testing environments.

8. Test Hooks: The framework provides hooks that allow the execution of specific code before or after certain events in the test lifecycle. For example, before_scenario and

after_scenario hooks can be used to set up preconditions or perform cleanup actions. Test hooks enhance the flexibility and maintainability of the framework.

9. Logging and Debugging: The framework incorporates logging mechanisms to capture relevant information during test execution. Logging helps in troubleshooting and provides valuable insights into the test flow, actions performed, and any errors encountered. Debugging capabilities enable developers and testers to identify and fix issues efficiently.

10. Test Dependencies and Parallel Execution: The framework supports managing dependencies between test scenarios to ensure proper sequencing and execution. It also enables parallel execution of tests, utilizing the capabilities of Selenium Grid or other parallel execution frameworks. Parallel execution helps save time and improves overall test efficiency.

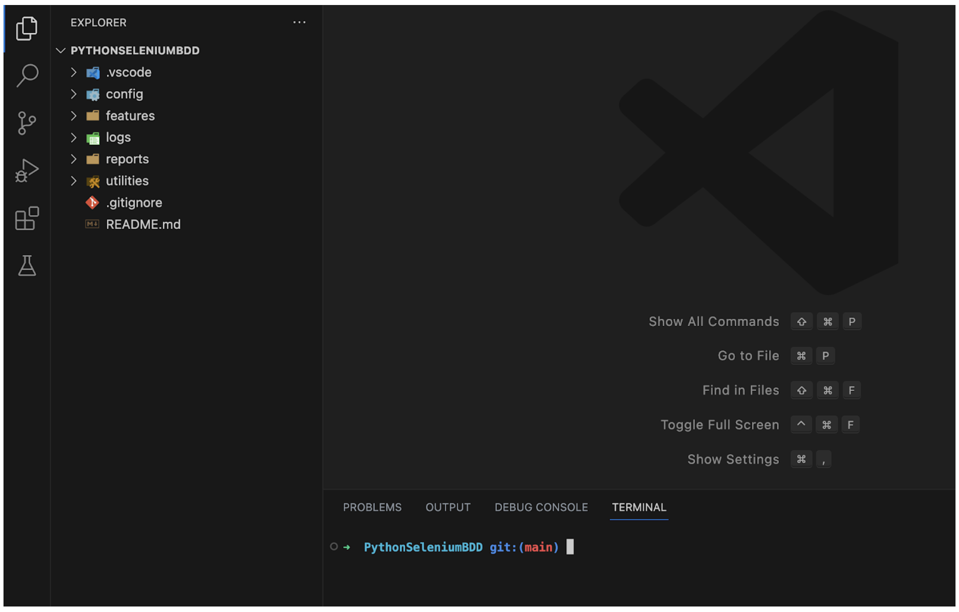
Automation testers won't necessarily be expected to create a framework from scratch but will have to be very familiar with creating and maintaining Feature files, Step Definitions and Page Objects. They will also need to know how to Run and Debug the tests as well as generating reports.

Let's get started with our own BDD framework

An example of a framework incorporating the majority of the above features can be cloned or downloaded from https://github.com/camsh69/Python-Selenium-BDD-Framework (click on the green '<> Code' button).

The framework is written in Python and uses a BDD tool called Behave to run the tests. The README.md file at the above link details all the steps required to get up and running.

Once you have cloned or downloaded and unzipped the code from GitHub, open the PythonSeleniumBDD folder in VS Code:

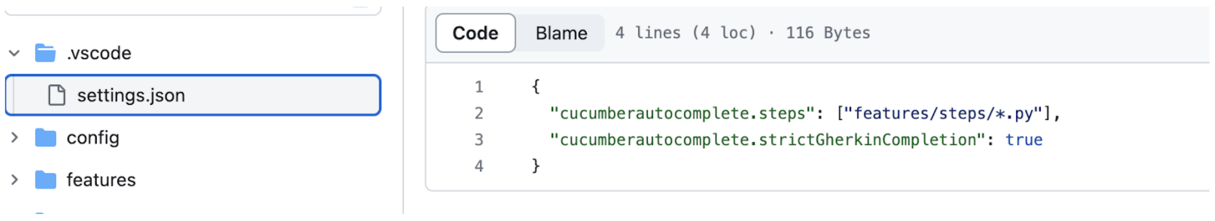


```
→ PythonSeleniumBDD git:(main)
```

Project structure and organization

Let’s take a look at the packages and modules in the framework and get an understanding of how it works.

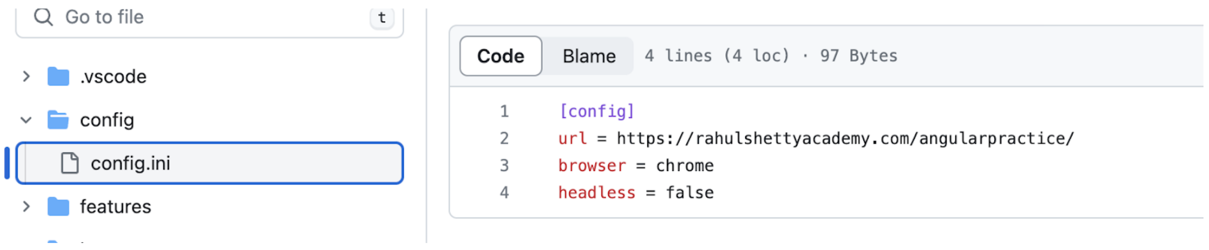
settings.json



```json
{
  "cucumberautocomplete.steps": ["features/steps/*.py"],
  "cucumberautocomplete.strictGherkinCompletion": true
}
```

The module settings.json contains some config if you are using VS Code as your code editor with the Cucumber (Gherkin) extension and allows you to click from a feature file step through to its step definition.

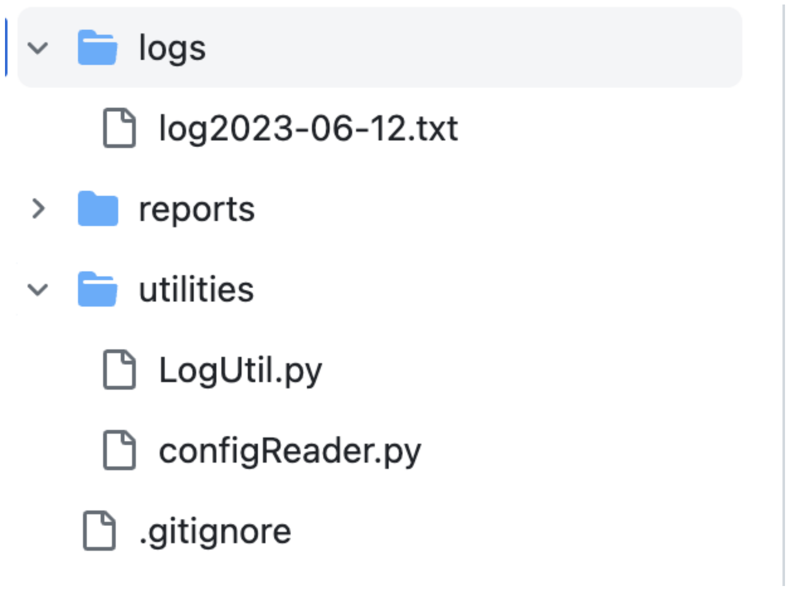
config.ini



```ini
[config]
url = https://rahulshettyacademy.com/angularpractice/
browser = chrome
headless = false
```

The config.ini module allows us to configure the URL under test, browser choice and whether we want to run chrome in headless mode or not. Set to false it will launch a browser on your machine, set to true the tests will run without a browser being launched.

logs, reports and utilities


```
logs
    log2023-06-12.txt
reports
utilities
    LogUtil.py
    configReader.py
.gitignore
```

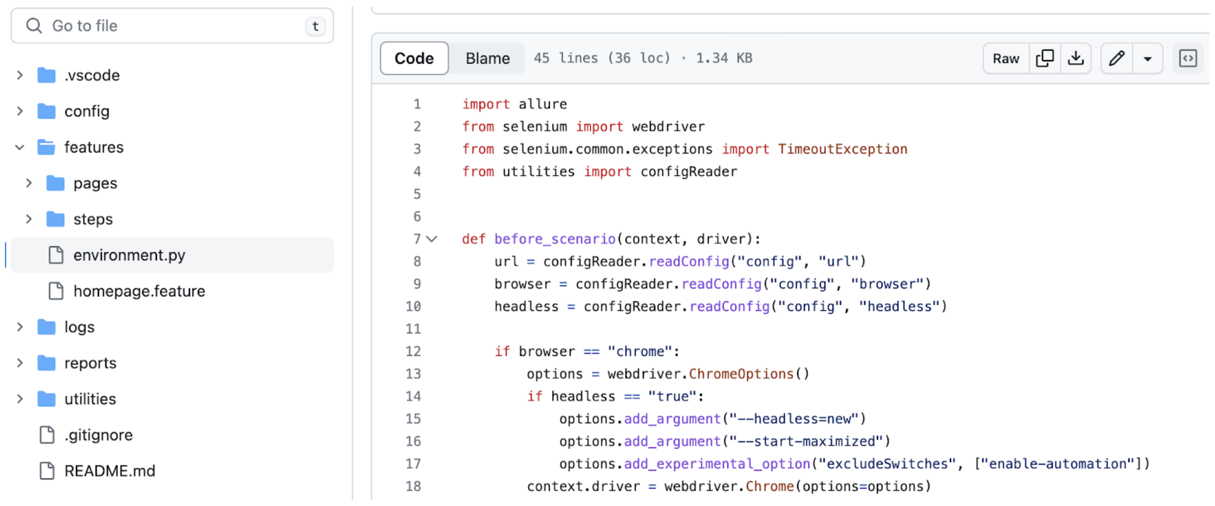
Let’s skip the features package for a moment and look at logs, reports and utilities. The logs package will contain a list of user defined entries for every test run and can be useful for debugging.

The reports package will contain the .json files and screenshots for populating an Allure report if we specify these to be captured (more on Allure in a later section).

The utilities package contains modules to make our logging and config.ini work.

Finally, .gitignore is a file we use if there is anything on our local machines we don’t want to or isn’t necessary to upload to GitHub (typically _pycache_ packages when we’re using python).

environment.py



```python
import allure
from selenium import webdriver
from selenium.common.exceptions import TimeoutException
from utilities import configReader


def before_scenario(context, driver):
    url = configReader.readConfig("config", "url")
    browser = configReader.readConfig("config", "browser")
    headless = configReader.readConfig("config", "headless")

    if browser == "chrome":
        options = webdriver.ChromeOptions()
        if headless == "true":
            options.add_argument("--headless=new")
            options.add_argument("--start-maximized")
            options.add_experimental_option("excludeSwitches", ["enable-automation"])
        context.driver = webdriver.Chrome(options=options)
```

Now let's take a look at the features package. The first module to look at is environment.py, there is nothing in this module that we need to manually change but it's helpful to know what it does.

It is how Behave uses hooks to determine what we want to happen at the start of every test scenario, the end of every test scenario and any special action we want to take after a specific step.

In our example, the before_scenario hook is taking the info from our config.ini to launch our chosen browser and navigate to our chosen URL at the start of our test.

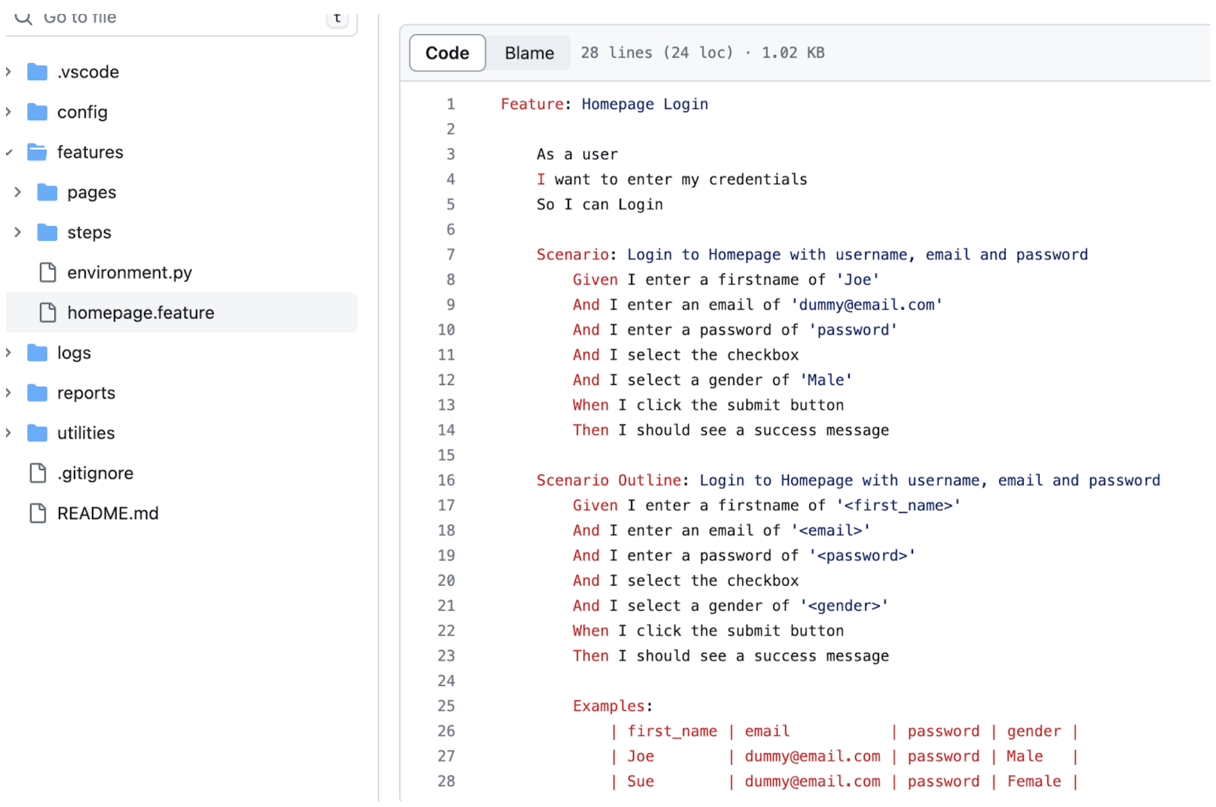
The after_scenario hook is quitting the browser after the test finishes. The after_step hook takes a screenshot if a step fails.

The other thing of note here is the 'context' keyword. In Behave, the `context` keyword refers to the context object that is passed between the steps in a scenario. It serves as a shared data container where you can store and access information during the execution of the scenario.

The `context` object is typically defined in the environment.py file, which acts as a bridge between the feature files and step definitions. It is automatically created and passed to each step definition method as an argument.

You can use the `context` object to store data or state that needs to be shared across steps within a scenario. For example, you can store the WebDriver instance in the `context` object, allowing you to access it from different steps to interact with the web application.
By leveraging the `context` object effectively, you can share information, manage state, and ensure proper coordination between steps in your BDD scenarios.

Feature Files



```gherkin
Feature: Homepage Login

    As a user
    I want to enter my credentials
    So I can Login

    Scenario: Login to Homepage with username, email and password
        Given I enter a firstname of 'Joe'
        And I enter an email of 'dummy@email.com'
        And I enter a password of 'password'
        And I select the checkbox
        And I select a gender of 'Male'
        When I click the submit button
        Then I should see a success message

    Scenario Outline: Login to Homepage with username, email and password
        Given I enter a firstname of '<first_name>'
        And I enter an email of '<email>'
        And I enter a password of '<password>'
        And I select the checkbox
        And I select a gender of '<gender>'
        When I click the submit button
        Then I should see a success message

        Examples:
            | first_name | email           | password | gender |
            | Joe        | dummy@email.com | password | Male   |
            | Sue        | dummy@email.com | password | Female |
```

Next, let’s turn our attention finally to how we as testers write Selenium tests in BDD.

Above is a feature file written in a typical way with Gherkin. The first example is a scenario where the data we want to pass to the web page under test is passed as a parameter in the actual step.

The second example is of a scenario outline that performs the same steps as example one but the data is passed in as parameters from an Examples table. Example two will run twice, once for first_name ‘Joe’ and then again for first_name ‘Sue’.

Writing effective feature files is crucial for successful BDD test automation. Here are some guidelines to help you write effective feature files:

1. Use Clear and Descriptive Titles: Start each feature file with a clear and descriptive title that captures the essence of the feature being tested. The title should convey the intended behaviour or functionality of the system.

2. Define a Feature Description: After the title, provide a brief description of the feature. This description should provide additional context and explain the purpose of the feature. It helps stakeholders, including developers and testers, to understand the feature's scope and expected outcomes.

3. Identify Scenarios: Scenarios represent specific test cases or examples that demonstrate the behaviour of the system. Each scenario should focus on a specific aspect of the feature being tested. Make sure the scenarios are concise, clear, and easy to understand. Use descriptive names for scenarios that capture the essence of the behaviour being tested.

4. Use Given-When-Then Structure: Follow the Given-When-Then structure to define the steps of each scenario. The Given step sets up the initial context or preconditions, the When step describes the action or event being performed, and the Then step defines the expected outcome or behaviour of the system. This structure provides clarity and improves the readability of the scenarios.

5. Keep Scenarios Independent: Ensure that each scenario is independent of others and can be executed in isolation. Avoid creating dependencies between scenarios, as it can lead to false positives or negatives in test results. Independent scenarios make it easier to understand failures and isolate issues.

6. Use Backgrounds and Scenario Outlines: Utilize Backgrounds to define common preconditions or steps that are applicable to multiple scenarios within a feature file. This helps in reducing redundancy and makes the feature file more concise. Additionally, Scenario Outlines can be used to define parameterized scenarios that can be executed with different sets of data.

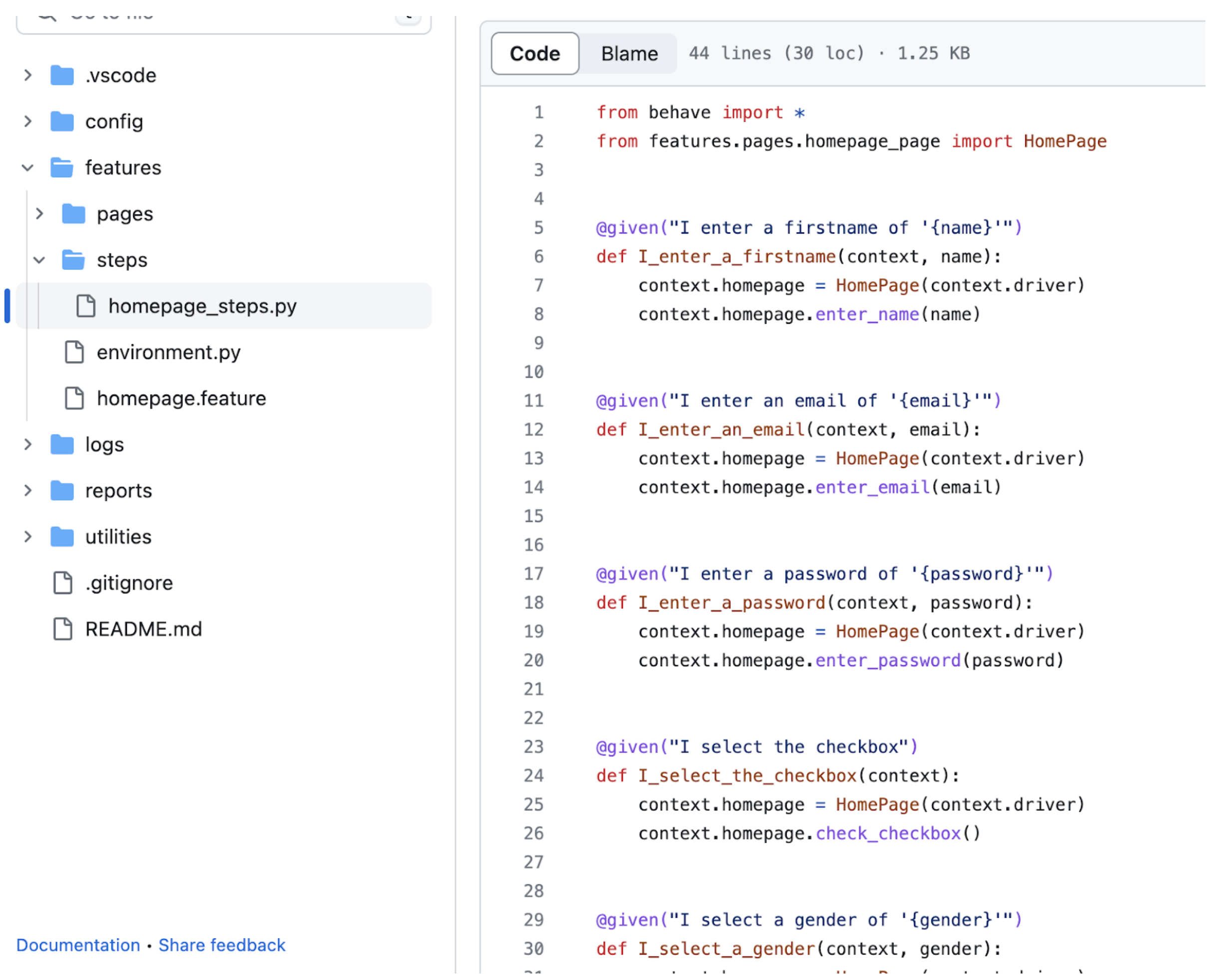
7. Write Readable Steps: Craft clear and readable step definitions that can be easily understood by all stakeholders. Use simple and concise language, avoid technical jargon, and focus on the desired behaviour rather than implementation details. Well-written step definitions improve collaboration and make it easier to review, maintain, and troubleshoot the test scenarios.

8. Add Comments and Annotations: Use comments and annotations in the feature file to provide additional context, explanations, or notes for future reference. Comments can be used to document decisions, assumptions, or any other relevant information related to the feature or scenarios.

9. Regularly Review and Refactor: Feature files should be periodically reviewed and refactored to ensure they remain up-to-date, relevant, and maintainable. As the system evolves, some scenarios may become obsolete or require modifications. Regular reviews help identify areas for improvement, eliminate redundancy, and enhance the overall quality of the feature files.

Learn more about Gherkin and Feature files here:
https://cucumber.io/docs/gherkin/reference/

Step Definitions



- .vscode
- config
- features
  - pages
  - steps
    - homepage_steps.py
  - environment.py
  - homepage.feature
- logs
- reports
- utilities
- .gitignore
- README.md

Documentation • Share feedback

```python
from behave import *
from features.pages.homepage_page import HomePage


@given("I enter a firstname of '{name}'")
def I_enter_a_firstname(context, name):
    context.homepage = HomePage(context.driver)
    context.homepage.enter_name(name)


@given("I enter an email of '{email}'")
def I_enter_an_email(context, email):
    context.homepage = HomePage(context.driver)
    context.homepage.enter_email(email)


@given("I enter a password of '{password}'")
def I_enter_a_password(context, password):
    context.homepage = HomePage(context.driver)
    context.homepage.enter_password(password)


@given("I select the checkbox")
def I_select_the_checkbox(context):
    context.homepage = HomePage(context.driver)
    context.homepage.check_checkbox()


@given("I select a gender of '{gender}'")
def I_select_a_gender(context, gender):
```

Now, let's look at the steps package and our Step Definition example.

In the context of BDD frameworks like Behave, a Step Definition is a Python function that maps to a step in a feature file. It is responsible for implementing the logic and actions associated with that step. Step Definitions allow you to define the behaviour or functionality for each step in a declarative manner.

A Step Definition is typically written using decorators provided by the BDD framework. For example, Behave provides the `given`, `when`, and `then` decorators to define step definitions for different types of steps. These decorators help in organizing and categorizing the steps based on their purpose.

Also note we need to import Behave at the top of the file and also the corresponding page object file (more to come on that soon).

Further reading on Behave can be found here:
Behave Documentation: https://behave.readthedocs.io/

Best practices for writing step definitions include:

1. Descriptive Step Definitions: Write step definitions that are clear, concise, and descriptive. Use meaningful names that reflect the intent of the step. This makes the scenarios more readable and understandable.

2. Reusability: Design step definitions to be reusable across scenarios. Avoid duplicating code or logic within step definitions. Extract common actions or functions into separate reusable methods or modules.

3. Keep Step Definitions Focused: Each step definition should focus on a single action or behaviour. Avoid including too many actions or complex logic within a single step definition. This promotes better maintainability and readability.

4. Use Page Objects: Encapsulate the interactions with web elements in Page Objects (more on this soon). Separate the element locators and interactions into dedicated Page Object classes. This helps in keeping step definitions clean and focused on the high-level actions rather than low-level element interactions.

5. Data Sharing with Context: Utilize the `context` object to share data between step definitions within a scenario. Store relevant data in the context object to pass it between steps. However, exercise caution not to abuse the context object and keep it limited to necessary shared data.

6. Error Handling and Assertions: Include appropriate error handling mechanisms in step definitions to handle exceptions or failures gracefully. Use assertions to verify the expected outcomes or states during the execution of steps.

7. Maintain Readability: Format and structure your step definitions for better readability. Use indentation, comments, and whitespace appropriately to make the code more organized and understandable.

Remember, step definitions play a crucial role in connecting the feature files with the underlying implementation. By following best practices and writing clear, reusable, and focused step definitions, you can create maintainable and effective BDD test automation scenarios.

Page Objects

Finally, we come to Page Objects. Before we look at the modules in the framework, let’s first get an understanding of the Page Object Model concept.

Page Objects is a design pattern commonly used in Selenium test automation to create an abstraction layer between the test code and the UI elements of a web application. It helps in organizing and managing the interactions with web elements by encapsulating them within dedicated Page Object classes.

The main idea behind Page Objects is to represent each page or component of a web application as a separate class. Each Page Object class contains the element locators and methods that represent the actions or interactions possible on that page/component. This separation of concerns makes the test code more maintainable, readable, and reusable.

Here are some best practices for implementing and using Page Objects effectively:

1. Single Responsibility Principle: Each Page Object class should have a single responsibility, representing a specific page or component. Avoid creating bloated or overloaded Page Objects that handle multiple unrelated pages.

2. Encapsulation: Hide the internal details of the page and expose only the necessary methods or actions that can be performed on that page. This provides a clean interface for the test code to interact with the page and promotes abstraction.

3. Element Locators: Encapsulate the element locators within the Page Objects. Avoid exposing the locators directly to the test code. This helps in decoupling the test code from the changes in the UI and makes it easier to update the locators when needed.

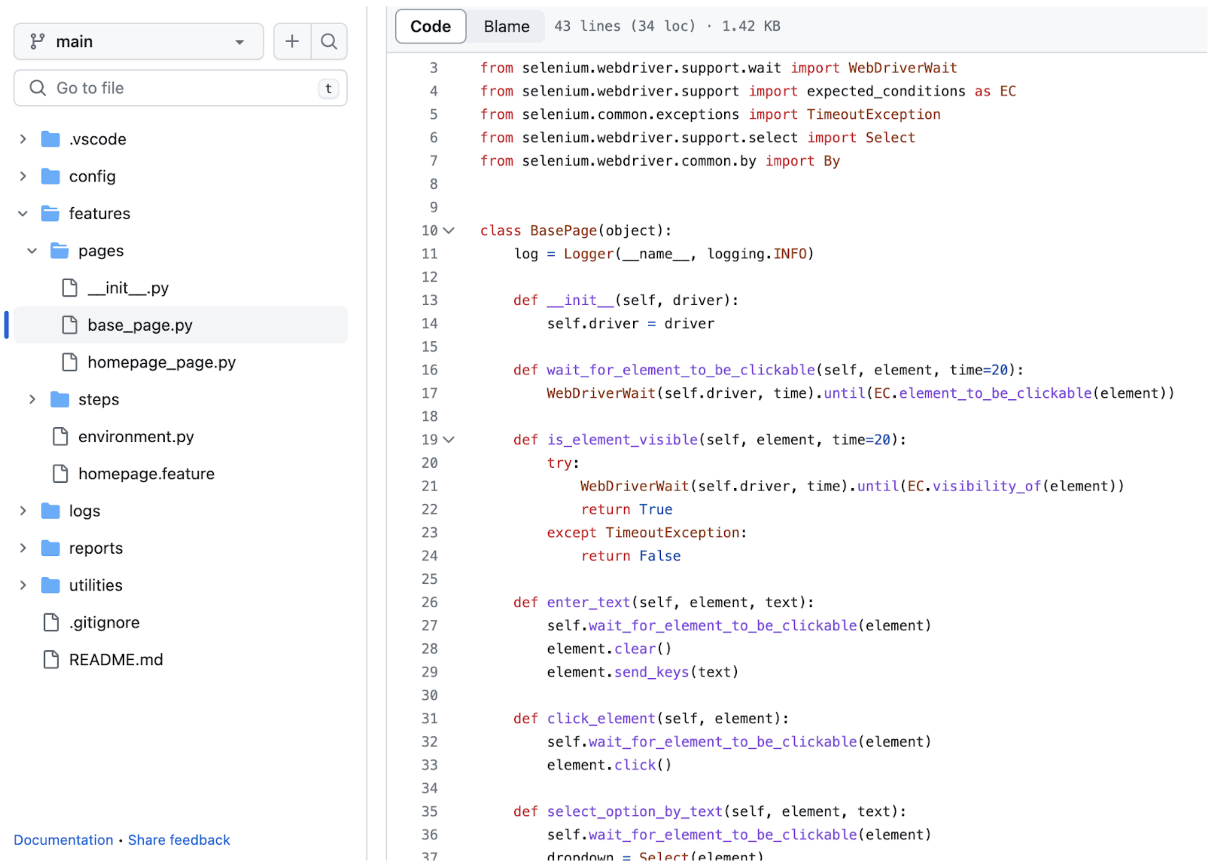
4. Action Methods: Create methods within the Page Objects that represent the actions or interactions possible on that page/component. These methods should abstract away the low-level details of element interactions and provide higher-level actions for the test code to use.

5. Reusability: Design the Page Objects to be reusable across multiple tests or scenarios. Avoid duplicating code or actions within the Page Objects. Extract common actions or interactions into separate reusable methods that can be called from multiple Page Objects.

6. Maintainability: Keep the Page Objects up to date with the changes in the application UI. Regularly review and update the locators and methods based on any UI changes. This ensures the stability and maintainability of the Page Objects over time.

7. Separation of Concerns: Separate the test logic and assertions from the Page Objects. The Page Objects should focus on providing methods for interacting with the UI, while the Step Definitions should handle the test-specific logic and assertions.

By following these best practices, Page Objects can greatly improve the structure, readability, and maintainability of your Selenium test automation code. They promote reusability, reduce code duplication, and provide a clear separation of concerns between the test code and the UI interactions.



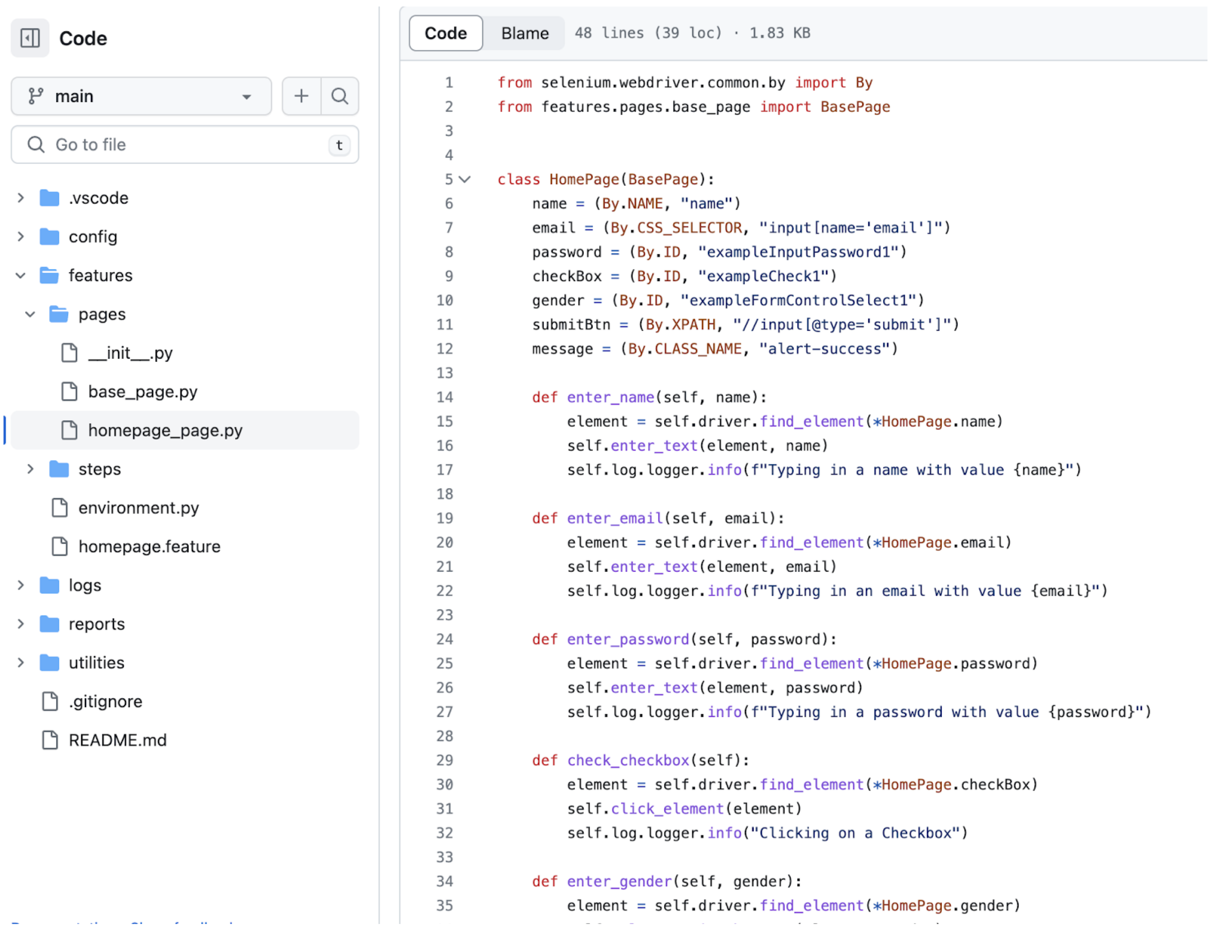
```python
from selenium.webdriver.support.wait import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.common.exceptions import TimeoutException
from selenium.webdriver.support.select import Select
from selenium.webdriver.common.by import By


class BasePage(object):
    log = Logger(__name__, logging.INFO)

    def __init__(self, driver):
        self.driver = driver

    def wait_for_element_to_be_clickable(self, element, time=20):
        WebDriverWait(self.driver, time).until(EC.element_to_be_clickable(element))

    def is_element_visible(self, element, time=20):
        try:
            WebDriverWait(self.driver, time).until(EC.visibility_of(element))
            return True
        except TimeoutException:
            return False

    def enter_text(self, element, text):
        self.wait_for_element_to_be_clickable(element)
        element.clear()
        element.send_keys(text)

    def click_element(self, element):
        self.wait_for_element_to_be_clickable(element)
        element.click()

    def select_option_by_text(self, element, text):
        self.wait_for_element_to_be_clickable(element)
        dropdown = Select(element)
```

Now we have a better understanding of what Page Objects are and why we use them, let's see how they've been implemented in our framework. First, let's look at base_page.py.

This module includes a parent class called BasePage, which serves as a foundation for all other page object modules. Inheriting from the BasePage class allows subsequent classes to access the variables and methods defined within it. This approach is beneficial because it enables us to write code once for commonly used functionalities, ensuring code reuse and avoiding repetition in our test automation code.

You'll notice a lot of the Selenium methods we looked at earlier and are likely to use often are here, e.g. **'click()'** , **'send_keys()'** and our wait methods.

Also note all the packages we need to import at the top of the page to get access to Selenium's functionality.



```python
from selenium.webdriver.common.by import By
from features.pages.base_page import BasePage


class HomePage(BasePage):
    name = (By.NAME, "name")
    email = (By.CSS_SELECTOR, "input[name='email']")
    password = (By.ID, "exampleInputPassword1")
    checkBox = (By.ID, "exampleCheck1")
    gender = (By.ID, "exampleFormControlSelect1")
    submitBtn = (By.XPATH, "//input[@type='submit']")
    message = (By.CLASS_NAME, "alert-success")

    def enter_name(self, name):
        element = self.driver.find_element(*HomePage.name)
        self.enter_text(element, name)
        self.log.logger.info(f"Typing in a name with value {name}")

    def enter_email(self, email):
        element = self.driver.find_element(*HomePage.email)
        self.enter_text(element, email)
        self.log.logger.info(f"Typing in an email with value {email}")

    def enter_password(self, password):
        element = self.driver.find_element(*HomePage.password)
        self.enter_text(element, password)
        self.log.logger.info(f"Typing in a password with value {password}")

    def check_checkbox(self):
        element = self.driver.find_element(*HomePage.checkBox)
        self.click_element(element)
        self.log.logger.info("Clicking on a Checkbox")

    def enter_gender(self, gender):
        element = self.driver.find_element(*HomePage.gender)
```

Last but not least, let's take a closer look at `homepage_page.py`. There are a few key points to note.

Firstly, the `HomePage` class is inheriting from the `BasePage` class. This inheritance allows the `HomePage` class to access and utilize all the methods and the logger defined in the `BasePage` class. This inheritance promotes code reuse and avoids redundancy by leveraging the existing functionality provided by the `BasePage` class.

Secondly, in the `HomePage` class, we define the locators that we have identified using tools like Chrome DevTools and SelectorsHub. These locators help us uniquely identify the elements on the web page that we want to interact with during our testing. By defining the locators within the page object class, we encapsulate the element identification logic, making it easier to maintain and update in the future.

Thirdly, if we recall our step definitions, they were calling the methods defined in our page object class, such as `enter_name(name)`. Within these methods, we can see that we are invoking methods from the `BasePage` class. This approach helps us avoid repetitive code by reusing the common functionality defined in the `BasePage` class. Additionally, it is considered good practice to include logging statements within our methods. Logging the steps we take during test execution can aid in debugging and provide valuable insights into the test flow.

To sum up, the `homepage_page.py` file showcases the utilization of inheritance from the `BasePage` class, the definition of locators to identify elements, and the usage of methods from the `BasePage` class to avoid code duplication. Including logging statements in methods further enhances debugging capabilities and provides a clear record of test execution steps.

As an aside, you may also have noticed the use of the keyword 'self' a lot in the code. In Python classes and methods, the `self` keyword is used to refer to the instance of the class itself. It is a convention in Python to use `self` as the first parameter in the method definition. When a method is called on an instance of a class, the instance is automatically passed as the `self` argument, allowing the method to access the attributes and methods of that specific instance.

The `self` parameter is used within the methods of a class to access and manipulate instance variables and call other methods of the class. It acts as a reference to the current instance, allowing the code to distinguish between the instance variables and methods of different objects of the same class.

By using `self`, we can maintain the state and identity of each instance of a class, ensuring that the correct instance variables and methods are accessed and modified within the scope of the class.

To sum up, `self` is a reference to the current instance of a class and is used to access instance variables and methods within class methods. It allows for proper encapsulation and differentiation between instances of the same class.

In summary

Don't worry too much if there seems to be a lot of technical detail and it all seems a bit overwhelming to take in. It'll be helpful in the long term if you understand all the concepts explained here but you'll get 95% of the way just following and copying the same patterns you see in the example Feature file, Step Definition and Page Objects.

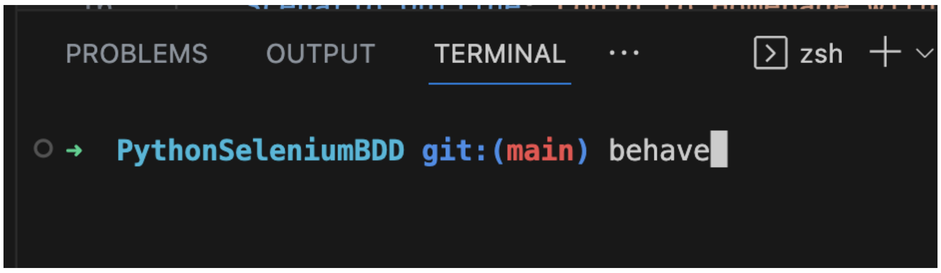
Different companies will all implement their frameworks slightly differently and different programming languages come with different syntaxes and BDD frameworks (e.g. Cucumber, Cucumber-JVM, SpecFlow). However, if they are following a BDD approach they'll most likely all be using Feature files, Step Definitions and Page Objects so there shouldn't be too many surprises.

The best way to learn is to jump in and start trying to automate web pages of your choice. Some sites to practise your Selenium and BDD skills include:

- https://www.selenium.dev/selenium/web/web-form.html
- https://selectorshub.com/xpath-practice-page/
- https://hub.testingtalks.com.au/playground

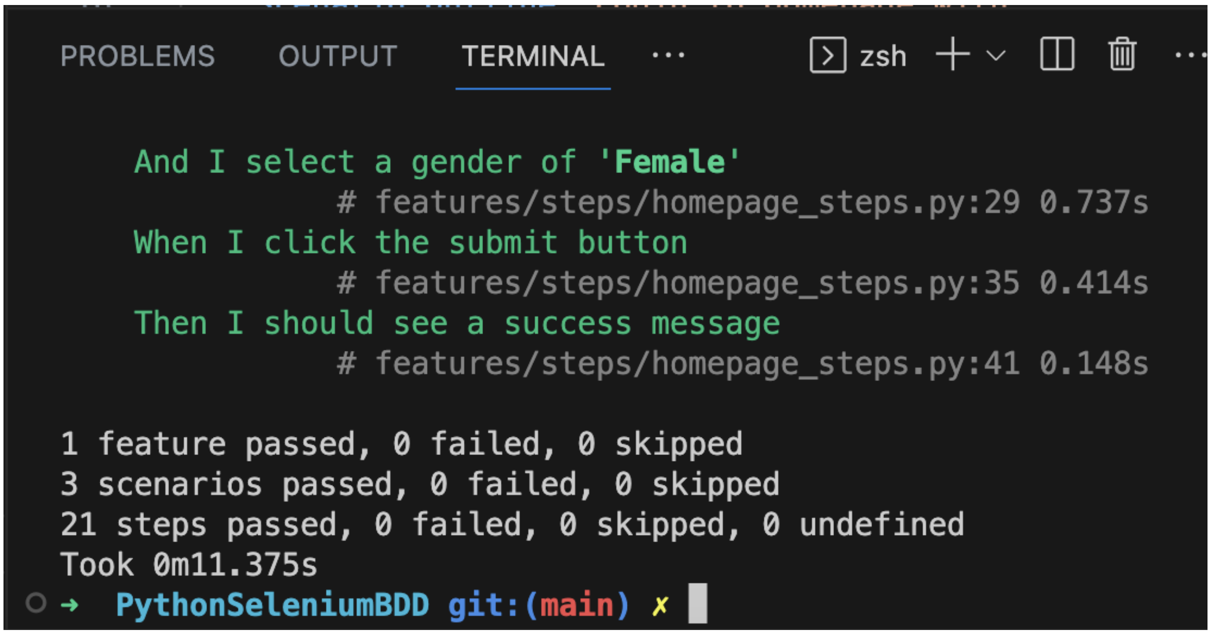
Running tests

The command to run the tests in your code editor's terminal is **'behave'**.



```
○ → PythonSeleniumBDD git:(main) behave
```

You'll see a browser launch (if not running headless) and Selenium interacting with the webpage under test. In your terminal, you should see your steps displayed as they either pass or fail and a summary of all pass/fails once your tests have ended.



```
    And I select a gender of 'Female'
                # features/steps/homepage_steps.py:29 0.737s
    When I click the submit button
                # features/steps/homepage_steps.py:35 0.414s
    Then I should see a success message
                # features/steps/homepage_steps.py:41 0.148s

1 feature passed, 0 failed, 0 skipped
3 scenarios passed, 0 failed, 0 skipped
21 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m11.375s
○ → PythonSeleniumBDD git:(main) ✗
```

To run only a specific feature the command is **'behave features/***name_of_feature***.feature'**.

It's also possible to run individual scenarios by using tagging. More details can be found in the Behave documentation: https://behave.readthedocs.io/en/latest/tag_expressions.html.

Debugging

The simple truth as an automation tester is that a chunk of your time will be spent debugging why your code isn't working as you are creating your test cases. This is normal and it is by first identifying and solving each problem as you come across it that you will succeed.

Here are some best practices to effectively debug Behave tests:

1. Enable Verbose Output: Behave provides a **`--verbose`** flag that can be added to the command line when running your tests. This flag enables more detailed output, including logging information and stack traces, which can help pinpoint the source of errors.

2. Use Print Statements: Inserting print statements within your step definitions or custom code can be a quick way to check the values of variables or identify specific points in the test execution flow. Print statements can provide valuable insights into the state of your test and help identify any unexpected behaviours.

3. Utilize Logging: Logging is an essential tool for debugging in Behave. Instead of relying solely on print statements, use the built-in Python `logging` module to log relevant information during test execution. You can add log messages at various stages of your tests, such as before and after critical steps, to track the flow and identify potential issues.

4. Review Stack Traces: When an error occurs during test execution, Behave provides a detailed stack trace that shows the sequence of function calls leading up to the error. Reviewing the stack trace can help identify the specific line of code or step definition that triggered the error, allowing you to focus your debugging efforts more effectively.

5. Use Breakpoints: If you're using an integrated development environment (IDE) with Behave, such as PyCharm or Visual Studio Code, you can set breakpoints in your step definitions or custom code. Breakpoints allow you to pause the execution of the tests at specific points, giving you the opportunity to inspect variables, step through the code, and understand the behaviour of your tests in more detail.

6. Data Printing and Visualization: Behave allows you to access test context and scenario data within your step definitions. You can print or visualize this data to gain insights into the current state of the test and verify if it matches your expectations. For example, you can print the content of data tables, scenario outlines, or custom context objects to ensure they are correctly populated.

7. Use --tags to Isolate Tests: Behave supports the use of tags to selectively run specific scenarios or features. You can leverage tags to isolate problematic scenarios and focus your debugging efforts on those specific tests. By running a subset of tests, you can narrow down the scope of investigation and accelerate the debugging process.

8. Debugging Tools: Python provides various debugging tools and utilities that can be integrated with Behave. Examples include the `pdb` debugger, which allows you to step through the code and inspect variables interactively, and the `ipdb` debugger, which provides similar functionality with additional features like tab completion.

Remember, effective debugging requires a systematic approach and a good understanding of the codebase and the behaviour of your tests. By applying these best practices and utilizing the available debugging tools, you can efficiently identify and resolve issues in your Behave tests.

Analysing test failures

Another important aspect of the Automation Tester’s role is analysing and resolving failed test cases. There are several reasons why a previously working test starts to fail. The common reasons are:

- The code has changed and you need to update the test accordingly (using stable locators like ID over Xpath helps lessen this occurring)
- The test is flaky, often because Selenium is trying to interact with an element that is not yet interactable on the DOM (best practice is to use explicit waits to wait until an element does become interactable)
- The code has changed and a genuine bug has been introduced – cool, your test has done its job.

Here are some other strategies you can use specifically focused on investigating failed tests:

1. Analyse Failure Messages: When a test fails, Behave provides detailed failure messages that can give you insights into the cause of the failure. Carefully review the failure message to understand the nature of the failure, including any error messages or assertions that didn't pass. The failure message often highlights the exact step or line of code that triggered the failure, allowing you to focus your investigation.

2. Review Logs and Output: If you have implemented logging within your Behave tests, review the logs generated during the failed test execution. Logs can provide additional information about the test's execution flow, intermediate results, or any unexpected behaviours. Look for any error messages, warnings, or inconsistencies that might help identify the root cause of the failure.

3. Reproduce the Failure: Try to reproduce the failed test locally in your development environment. This allows you to debug the test interactively and observe the behaviour in real-time. By reproducing the failure, you can apply debugging techniques mentioned earlier, such as enabling verbose output, using print statements, or setting breakpoints to inspect variables and step through the code.

4. Narrow Down the Failure Scope: If you're dealing with a large feature or scenario that fails, it can be helpful to narrow down the scope of the investigation. Disable or skip unrelated steps or features to isolate the failure and focus on the specific section of code or step definitions that are causing the issue. By narrowing down the scope, you can streamline the investigation process and save time.

5. Analyse Test Data and Preconditions: Failed tests can sometimes be a result of incorrect test data or preconditions. Verify the input data and preconditions for the failed test scenario to ensure they are correct and properly set up. Check the values of variables, data tables, or external dependencies that might impact the test's outcome. By validating the test data and preconditions, you can identify any discrepancies that might be causing the failure.

6. Collaborate with the Team: If you're unable to identify the cause of the failure on your own, don't hesitate to seek help from your team members. Discuss the failure with developers, QA engineers, or other stakeholders involved in the project. Sometimes, a fresh pair of eyes or a different perspective can uncover potential issues or provide insights into resolving the failure.

7. Use Version Control: If your test code is managed using version control, such as Git, utilize its features to help investigate failed tests. Use the diff functionality to compare the current version of the test code with the previous working version. This can help identify any recent changes that might have introduced the failure or led to unexpected behaviour.

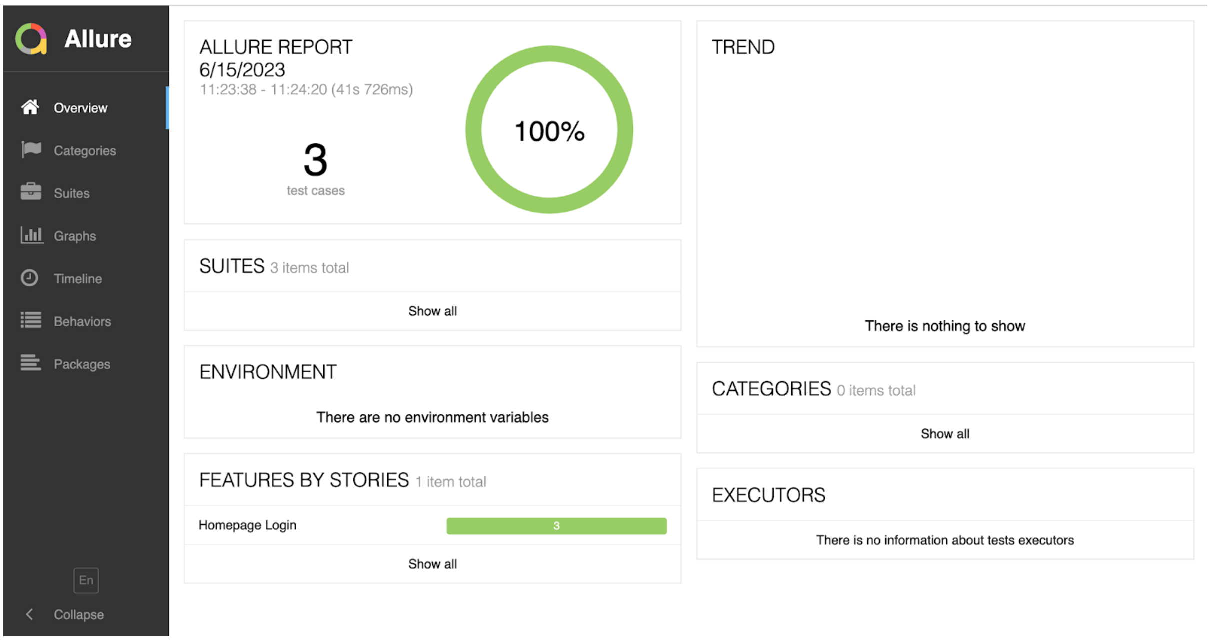
Remember, investigating failed tests requires a systematic approach and attention to detail. By analysing failure messages, reviewing logs, reproducing failures, narrowing down the scope, validating test data, collaborating with the team, and leveraging version control, you can effectively investigate and resolve issues in your Behave tests.

Generating reports

The framework includes the functionality to produce Allure reports

The command to run the tests and generate the files required for a report is **'behave -f allure_behave.formatter:AllureFormatter -o reports/ features'**.

To then generate a report, enter **'allure serve reports/'** into your terminal.



ALLURE REPORT 6/15/2023
11:23:38 - 11:24:20 (41s 726ms)
3
test cases
100%

SUITES 3 items total
Show all

ENVIRONMENT
There are no environment variables

FEATURES BY STORIES 1 item total
Homepage Login 3
Show all

TREND
There is nothing to show

CATEGORIES 0 items total
Show all

EXECUTORS
There is no information about tests executors

Allure is a powerful reporting framework that can be integrated with Behave to generate rich and interactive test reports. Allure provides detailed and visually appealing reports that enhance the visibility and understandability of test results.
When using Allure with Behave, you can generate reports that include information about your feature files, scenarios, and steps, along with additional details such as execution time, status (passed/failed), attachments (screenshots or other files), and log messages.

Some key features and benefits of Allure reports for Behave include:

1. Interactive Visualization: Allure reports offer a user-friendly and interactive interface that allows you to navigate through the test results easily. You can view detailed information about individual scenarios and steps, including their statuses and associated attachments.

2. Detailed Test Execution Statistics: Allure provides comprehensive statistics about test execution, such as the total number of tests, passed tests, failed tests, and their respective percentages. This helps you quickly assess the overall test coverage and success rate.

3. Rich Attachments: You can attach files like screenshots, log files, or any other relevant artifacts to the Allure report. This allows you to provide additional context and evidence for test failures or to capture important information during test execution.

4. History and Trend Analysis: Allure keeps a history of test runs, allowing you to compare results across multiple test executions. This enables trend analysis and helps identify patterns or recurring issues over time.

Here are some online resources to explore further about Allure with Behave:
- Allure official website: https://docs.qameta.io/allure/
- Allure Behave documentation: https://docs.qameta.io/allure/#_behave

By integrating Allure with Behave, you can enhance your test reporting capabilities and improve the visibility and clarity of your test results.

In conclusion

Congratulations, you now have all the resources, tools and best practices you need to become a highly skilled Automation Tester and provide real value to your automation projects.

Practise, practise, practise, trial and error and making lots of mistakes are the best ways to learn so please stick at it. It can be hugely frustrating at times if you can't get something to work but a real sense of achievement when you finally do – remember it was/still is the same for all of us.

It's also likely that you're not the first person to come across any problems that you find and often a search on Google, Stack Overflow or asking ChatGPT will provide you with that breakthrough you were looking for.

Good luck.