**WebDriver in selenium**

Selenium WebDriver is a popular web-based automation testing framework that is primarily used for automating tasks related to Web UI testing.

Selenium WebDriver does not interact directly with the web elements on a page. A browser-specific Selenium WebDriver acts as the bridge between the test script and the web browser.

[Selenium locators](https://www.lambdatest.com/learning-hub/selenium-locators) are used for locating elements on the page so that appropriate methods can be used for interacting with the element.

**Methods of selenium webdriver class**

|  |  |  |
| --- | --- | --- |
| **Method** | **Syntax** | **Description** |
| get() | driver.get(url) |
| current\_url(); | driver.current\_url() |
| getTitle() | driver.getTitle() |
| findElements() | driver.findElements(By by) |
| findElement() | driver.findElement(By by) |
| [add\_cookie](https://www.geeksforgeeks.org/add_cookie-driver-method-selenium-python/) |  | Adds a cookie to your current session. | |
| [back](https://www.geeksforgeeks.org/back-driver-method-selenium-python/?ref=rp) | driver.back() | Goes one step backward in the browser history. | |
| [close](https://www.geeksforgeeks.org/close-driver-method-selenium-python/?ref=rp) | driver.close() | Closes the current window. | |
| [create\_web\_element](https://www.geeksforgeeks.org/create_web_element-driver-method-selenium-python/?ref=rp) |  | Creates a web element with the specified element\_id. | |
| [delete\_all\_cookies](https://www.geeksforgeeks.org/delete_all_cookies-driver-method-selenium-python/?ref=rp) |  | Delete all cookies in the scope of the session. | |
| [delete\_cookie](https://www.geeksforgeeks.org/delete_cookie-driver-method-selenium-python/?ref=rp) |  | Deletes a single cookie with the given name. | |
| [delete\_cookie](https://www.geeksforgeeks.org/delete_cookie-driver-method-selenium-python/?ref=rp)  [execute\_async\_script](https://www.geeksforgeeks.org/execute_async_script-driver-method-selenium-python/) |  | Deletes a single cookie with the given name.  Asynchronously Executes JavaScript in the current window/frame. | |
|  |
| [execute\_script](https://geeksforgeeks.org/execute_script-driver-method-selenium-python/) |  | Synchronously Executes JavaScript in the current window/frame. | |
| [forward](https://www.geeksforgeeks.org/forward-driver-method-selenium-python/) |  | Goes one step forward in the browser history. | |
| [fullscreen\_window](https://www.geeksforgeeks.org/fullscreen_window-driver-method-selenium-python/) |  | Invokes the window manager-specific ‘full screen’ operation | |
| [get\_cookie](https://www.geeksforgeeks.org/add_cookie-driver-method-selenium-python/?ref=rp) |  | Get a single cookie by name. Returns the cookie if found, None if not. | |
| [get\_cookies](https://www.geeksforgeeks.org/get_cookies-driver-method-selenium-python/?ref=rp) |  | Returns a set of dictionaries, corresponding to cookies visible in the current session. | |
| [get\_log](https://www.geeksforgeeks.org/get_log-driver-method-selenium-python/?ref=rp) | driver.get\_log(log\_type) | Gets the log for a given log type | |
| [get\_screenshot\_as\_base64](https://www.geeksforgeeks.org/get_screenshot_as_base64-driver-method-selenium-python/?ref=rp) |  | Gets the screenshot of the current window as a base64 encoded string which is useful in embedded images in HTML. | |
| [get\_screenshot\_as\_file](https://www.geeksforgeeks.org/get_screenshot_as_file-driver-method-selenium-python/?ref=rp) |  | Saves a screenshot of the current window to a PNG image file. | |
| [get\_screenshot\_as\_png](https://www.geeksforgeeks.org/get_screenshot_as_png-driver-method-selenium-python/?ref=rp) |  | Gets the screenshot of the current window as a binary data. | |
| [get\_window\_position](https://www.geeksforgeeks.org/get_window_position-driver-method-selenium-python/?ref=rp) |  | Gets the x, y position of the current window. | |
| [get\_window\_rect](https://www.geeksforgeeks.org/get_window_rect-driver-method-selenium-python/?ref=rp) |  | Gets the x, y coordinates of the window as well as height and width of the current window. | |
| [get\_window\_rect](https://www.geeksforgeeks.org/get_window_rect-driver-method-selenium-python/?ref=rp)  [get\_window\_size](https://www.geeksforgeeks.org/get_window_size-driver-method-selenium-python/) |  | Gets the x, y coordinates of the window as well as height and width of the current window.  Gets the width and height of the current window. | |
|  |
| [implicitly\_wait](https://www.geeksforgeeks.org/implicitly_wait-driver-method-selenium-python/?ref=rp) |  | Sets a sticky timeout to implicitly wait for an element to be found, | |
| [maximize\_window](https://www.geeksforgeeks.org/maximize_window-driver-method-selenium-python/?ref=rp) |  | Maximizes the current window that webdriver is using | |
| [minimize\_window](https://www.geeksforgeeks.org/minimize_window-driver-method-selenium-python/?ref=rp) |  | Invokes the window manager-specific ‘minimize’ operation | |
| [quit](https://www.geeksforgeeks.org/quit-driver-method-selenium-python/) | driver.quit() | Quits the driver and closes every associated window. | |
| [refresh](https://www.geeksforgeeks.org/refresh-driver-method-selenium-python/) |  | Refreshes the current page. | |
| [set\_page\_load\_timeout](https://www.geeksforgeeks.org/set_page_load_timeout-driver-method-selenium-python/?ref=rp) |  | Set the amount of time to wait for a page load to complete before throwing an error. | |
| [set\_script\_timeout](https://www.geeksforgeeks.org/set_script_timeout-driver-method-selenium-python/?ref=rp) | driver.set\_page\_load\_timeout(time\_To\_wait) | Set the amount of time that the script should wait during an execute\_async\_script call before throwing an error. | |
| [set\_window\_position](https://geeksforgeeks.org/set_window_position-driver-method-selenium-python/) |  | Sets the x, y position of the current window. (window.moveTo) | |
| [set\_window\_rect](https://geeksforgeeks.org/set_window_rect-driver-method-selenium-python/) |  | Sets the x, y coordinates of the window as well as height and width of the current window. | |
| [current\_url](https://geeksforgeeks.org/current_url-driver-method-selenium-python/) |  | Gets the URL of the current page. | |
| [current\_window\_handle](https://geeksforgeeks.org/current_window_handle-driver-method-selenium-python/) |  | Returns the handle of the current window. | |
| [page\_source](https://geeksforgeeks.org/page_source-driver-method-selenium-python/) |  | Gets the source of the current page. | |
| [title](https://geeksforgeeks.org/title-driver-method-selenium-python/) | driver.title | Returns the title of the current page. | |
| text | Web\_element.text | Returns the text of webelement | |

#############################################################################

# Action class in selenium #

#############################################################################

Actions class is an ability provided by Selenium for handling keyboard and mouse events. In Selenium WebDriver, handling these events includes operations such as drag and drop, clicking on multiple elements with the control key, among others.

Action can be divided into two parts-

* Mouse action
* Keyboard action

**Mouse actions in selenium:**

<https://www.selenium.dev/documentation/webdriver/actions_api/mouse/>

* double\_click(clickable): Performs double click on the element
* click\_and\_hold(clickable): Performs long click on the mouse without releasing it
* drag\_and\_drop(dragable): Drags the element from one point and drops to another
* move\_to\_element(movable): Shifts the mouse pointer to the centre of the element
* drag\_and\_drop\_by\_offset(source, x\_offset, y\_offset) --- move source to given coordinate
* context\_click(clickable): Performs right-click on the mouse
* click(clickable) : Click on clickable webelement

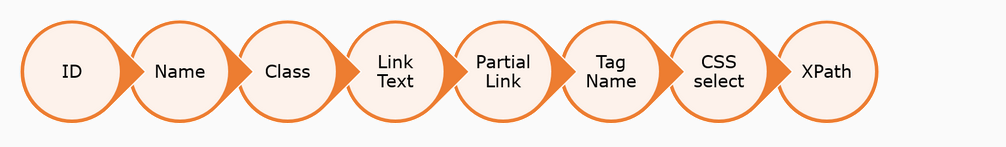
**Keyboard action in selenium**

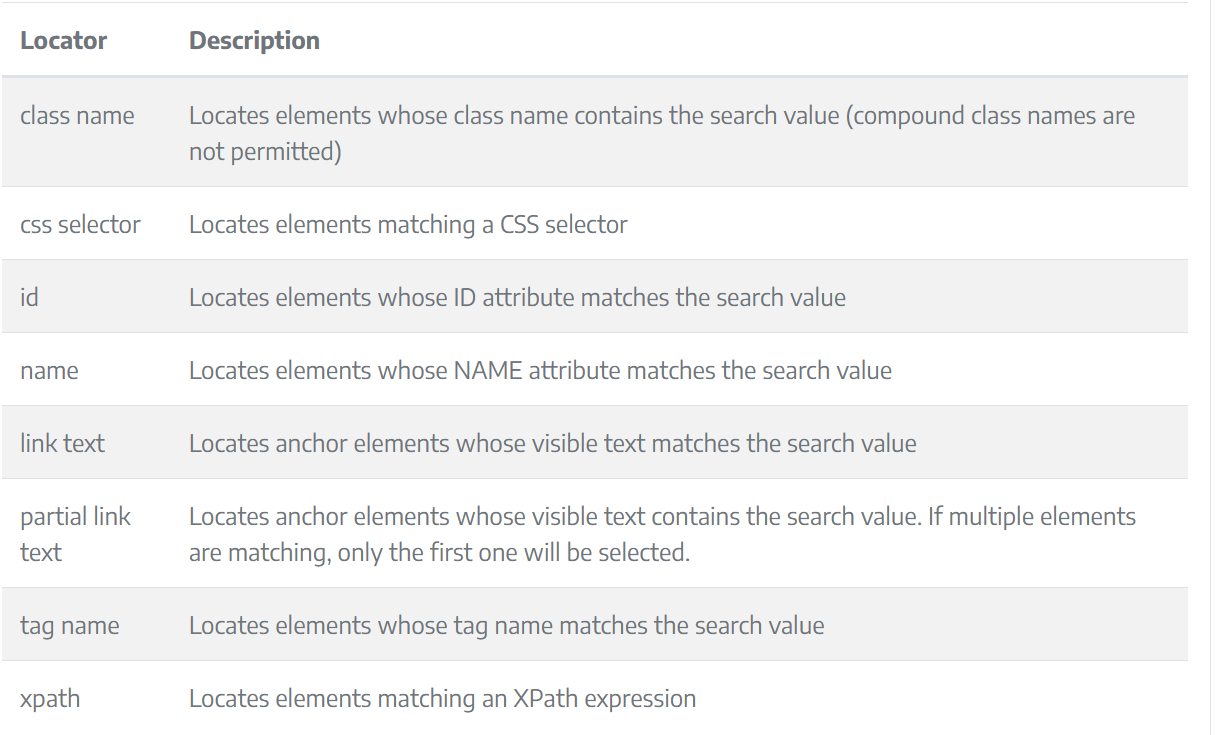
* send\_keys(): Sends a series of keys to the element
* key\_up(): Performs key release
* key\_down(): Performs keypress without release

################################################################################# Elementslocator or Locators in Selenium #

################################################################################

Selenium offers 8 types of locators as follows, which help in uniquely identifying an element in a web page.





**Name Select:**

Name select is used to select the web element using name attributes of web elements.

**Example:**

Consider the below html code. We want to locate the input field. Write code for it



Solution:

We can locate it using below line-

driver.find\_element(By.NAME,'loginName')

**Class name locator:**

It is used to locate the element using class name. They basically deal with the styling options added to the element

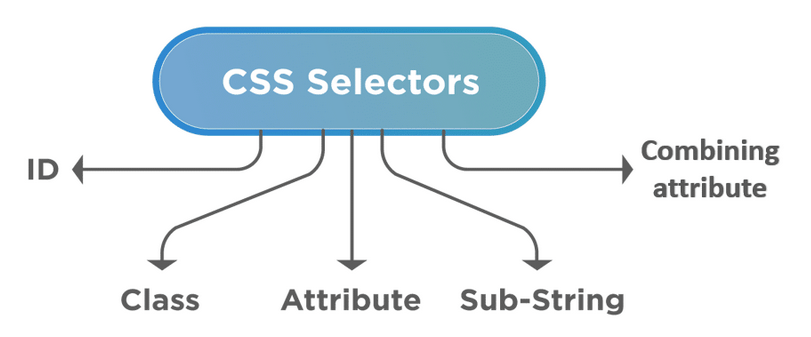
driver.findElement(By.className, (<element class>))

**#CSS Selector #**

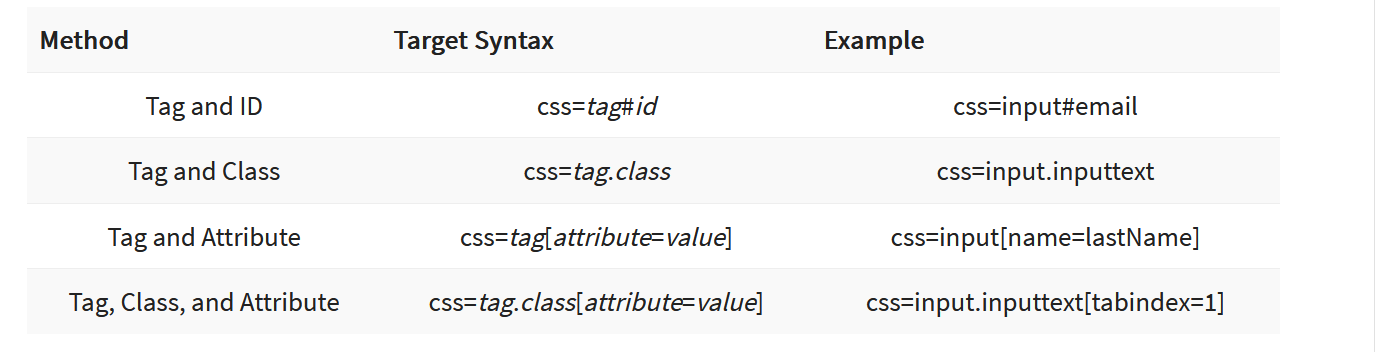
CSS Selectors in Selenium are string patterns used to identify an element based on a combination of HTML tag, id, class, and attributes. Locating by CSS Selectors in Selenium is more complicated.

If the elements you are trying to work with do not have unique id or class attributes associated with them, CSS selector is the saviour.

As shown by the following figure, CSS Selectors can be created based on the following strategies:



**Creating CSS Selector:**



**How to create a CSS selector**

We can create CSS selector in below way-

node[attribute\_name = ‘attribute\_value’] ----------- Method 1

* node is the tag name of the HTML element, which needs to locate.
* attribute\_name is the name of the attribute which can locate the element.
* attribute\_value is the value of the attribute, which can locate the element.

**CSS selector with id**

We can use CSS class and ID in below way to locate an element

node[attribute\_name = ‘attribute\_value’] ----------- Method 1

tag#id ----------- Method 2

**Example**: Frome below page locate the first name input text and enter your name. HTML is -

<input required="" autocomplete="off" placeholder="First Name" type="text" id="firstName" class=" mr-sm-2 form-control">

Website-

AUT='https://demoqa.com/automation-practice-form'

Solution:

AUT='https://demoqa.com/automation-practice-form'

driver=webdriver.Firefox(executable\_path='geckodriver.exe')

driver.get(AUT)

driver.find\_element(By.CSS\_SELECTOR,"input[id='firstName']").send\_keys('Rakesh')

#driver.find\_element(By.CSS\_SELECTOR,'input#firstName').send\_keys('Rakesh')

sleep(3)

driver.close()

driver.quit()

**CSS Selector using Class**

We can use CSS selector to locate element using class in below way-

node[attribute\_name = ‘attribute\_value’] ----------- Method 1

tag.class ----------- Method 2

**Question:**

For same application which we tested above locate the text area and enter some text. It’s HTML is -

<textarea placeholder="Current Address" rows="5" cols="20" id="currentAddress" class="form-control"></textarea>

**Solution:**

AUT='https://demoqa.com/automation-practice-form'

driver=webdriver.Firefox(executable\_path='geckodriver.exe')

driver.get(AUT)

driver.find\_element(By.CSS\_SELECTOR,"textarea[placeholder='CurrentAdress']").send\_keys('Rakesh')

#driver.find\_element(By.CSS\_SELECTOR,'textarea#form-control').send\_keys('Rakesh')

sleep(3)

driver.close()

driver.quit()

**CSS Selector with tag and attribute:**

css=tag[attribute=value] ----- Syntax

**CSS Selector with tag, class, and attribute**

css=tag.class[attribute=value]

**CSS Selector with inner text**

css=tag:contains("inner text")

**# XPath #**

Xpath – this is used to locate the element using relative path or full path in html or xml. XPath can be written in below way-

//tagname[@attribute='value']

//\*[@attribute='value']

XPath method can locate the element if we give full qualified path of any element.

If XPath starts with // then it will be considered as relative path

If XPath start with / then it will be considered as absolute path.

Example: Let us suppose we want to locate the form element

<html>

<body>

<form id="loginForm">

<input name="username" type="text" />

<input name="password" type="password" />

<input name="continue" type="submit" value="Login" />

<input name="continue" type="button" value="Clear" />

</form>

</body>

</html>

We can locate form element in below way-

1. login\_form = driver.find\_element(By.XPATH, "/html/body/form[1]")

----Absolute path (would break if the HTML was changed only slightly)

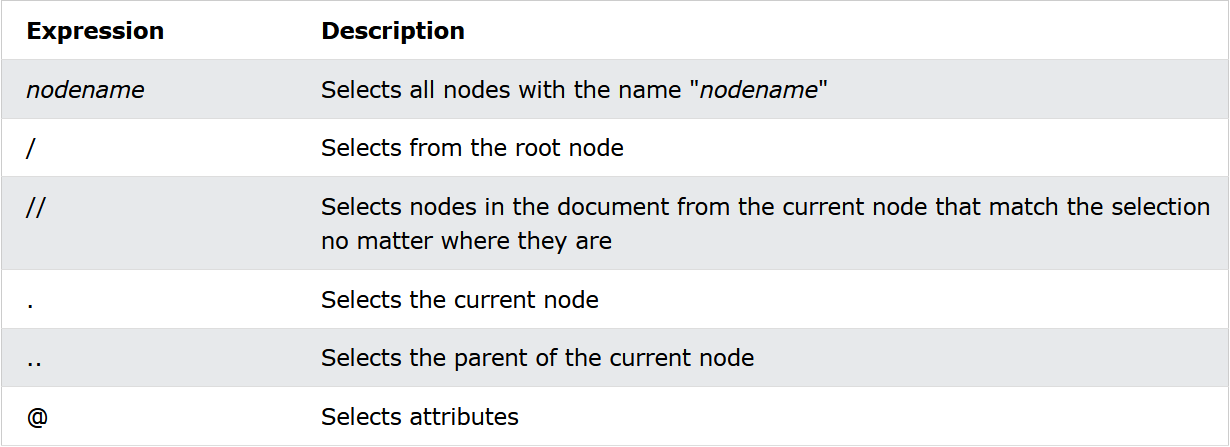
1. login\_form = driver.find\_element(By.XPATH, "//form[1]")

---- First form element in the HTML

1. login\_form = driver.find\_element(By.XPATH, "//form[@id='loginForm']")

------ The form element with attribute id set to loginForm

**Some more detail/used terms in xpath**



xpath[n] ------ it gives the nth element from all matches of xpath. n>=1

**Example on how to use xpath terms:**

|  |  |
| --- | --- |
| **Path Expression** | **Result** |
| Bookstore | Selects all nodes with the name "bookstore" |
| /bookstore | Selects the root element bookstore  **Note:** If the path starts with a slash ( / ) it always represents an absolute path to an element! |
| bookstore/book | Selects all book elements that are children of bookstore |
| //book | Selects all book elements no matter where they are in the document |
| bookstore//book | Selects all book elements that are descendant of the bookstore element, no matter where they are under the bookstore element |
| //@lang | Selects all attributes that are named lang |

**How To Write Dynamic XPath In Selenium WebDriver**

We can write XPath in below ways-

1. **Basic XPath**

In basic XPath expression it select nodes or list of nodes on the basis of attributes like ID , Name, Classname etc.

1. **Contains()**

Contains() is a method used in XPath expression. It is used when the value of any attribute changes dynamically, for example, login information.

It finds all web elements that contains/matches the given string/text

//tagname[contains(@attribute,'value')]

//\*[contains(@attribute,'value')]

**Example: ----- Good**

Xpath=//\*[contains(@type,**'sub'**)]

This will find any element for which type attribute value contains with ‘sub’

1. **Using OR & AND:**

It is used for selecting statement by applying logical operation.

//\*[@attrbute1='value1' or @attribute2='value2']

//tagname[@attrbute1='value1' or @attribute2='value2']

Example:

Xpath=//\*[@type='submit' or @name='btnReset']

**Example 2:**

Let us suppose we have below html data, and want to create x path.

<html>

<head>

<title>title one</title>

</head>

<body>

<h2>this is h2 tag with anchor <a href="#">Link</a></h2>

<p>I am first para here<p>

<!—some coment -->

</body>

</html>

**Question 1**: Creating a path for the "<title>" element.

Solution: //title - searches for the title element, it can appear anywhere in the document.

**Question 2**: Creating a path for the "<a>" anchor element.

Solution 1: //a -searches for the anchor element, it can appear anywhere in the document.

Solution 2: //h2/a -searches for the h2 element followed by the anchor tag, it can appear anywhere in the document.

**Question 3**: Creating a path for the "<p>" paragraph element.

Solution: //p -searches for the paragraph element, it can appear anywhere in the document

Example 3:

Let us suppose in a HTML page there are more than one anchor tag and want to select the first anchor tag using xpath.

Solution- //a[1] - would match only the first anchor tag

**Methods to find the elements:**



#ID locator#

Selenium uses the id attribute of an HTML element for locating a specific element.

The id of an element should be unique within a page, so the id selector is used to select one unique element.

ID locator in Selenium is the most preferred and fastest way to locate desired WebElements on the page. ID Selenium locators are unique for each element in the DOM.

#Name locator#

The name attribute specifies a name for the element. This name attribute can be used to locate the element in a web page.

# Class name #

This locator locates the element which matches the value specified in the attribute name “class”.They basically deal with the styling options added to the element (They might be inbuilt styles or customized styles added by the developer).

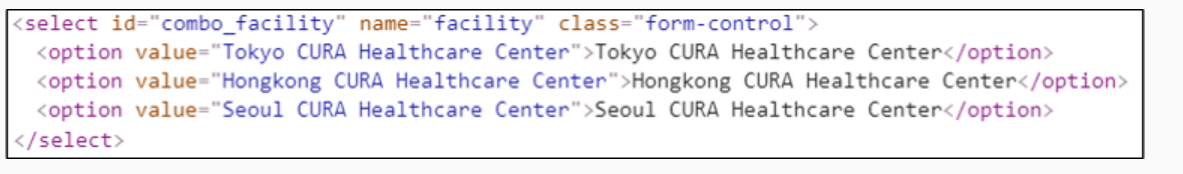
In the image given here, class= "table table-hover" is added to the table.

# Tag name #

This locator is used to find the elements matching the specified tag name.

This locator can be **used whenever there is no ID or Class Name or Name.** **Also when there is a need to locate an element and extract the elements nested within them.**

In the below image, suppose you want all the options to be extracted, tag name option can be used.



**# Partial Link text #**

In certain cases, we may need to find links by a portion of the text in a Link Text element. In such situations, we can use Partial Link Text to locate elements.

**Alternate way to search element**

Alternatively, we can search element in below way-



################################################################################

# WebElements in selenium #

################################################################################

Different types of elements are as follows-

1. Text Box
2. Link
3. Button
4. Image, image link, image button
5. Text area
6. Checkbox
7. Radio button
8. Dropdown list

**Interaction with web element**:

Suppose we want to click on a radio button, click a link, enter value in text box for this we need to select the element and then do the action. For this we have below method –

|  |  |
| --- | --- |
| find\_element(locator, value) | Returns the first webelement, matching the given locator and throws an exception if such an element is not found. |
| find\_elements(locator, value) | Returns the ll webelements, matching the given locator and throws an exception if such an element is not found. |

#####################################

# Working with radio button and check box #

#####################################

**How to locate radio buttons and checkbox:**

Locating of radio buttons or checkbox is same as other web elements, using css selector or id selector or x path based on the html page.

**Checking radio buttons and checkbox is state:**

When we are working with webpages sometimes, we need to verify that radio button or checkbox are selected or not, for this we have below methods in class BaseWebElement of selenium.

element.is\_displayed() → bool

Whether the element is visible to a user

element .is\_selected() → bool

Returns whether the element is selected.

element .is\_enabled() → bool

Returns whether the element is enabled.

**Note:**

We have get\_attribute("checked") that can also be used to check the status of radio button.

web\_element. get\_attribute("checked") ----- Way to use it for radio and checkbox

**Operation that can be performed on radio button and checkbox:**

* Select the radio button.
* Validate whether the radio button is selected or not.

**More on radio buttons:**

Radio buttons are special multi-valued fields that allow only one value among a group of values to be selected.

The radio button does not support deselection. To deselect a radio button, one needs to select any other radio button in that group.

**Key notes checkbox:**

It is special multi-valued fields that allow only one value among a group of values to be selected.

**Questions**:

Open the below link, accept cookies and select any radio button. After selecting radio button validate that radio button is selected.

<https://www.sugarcrm.com/au/request-demo/>

**Solution**:

from unicodedata import name

from selenium import webdriver

from selenium.webdriver.common.by import By

from time import sleep

import unittest

AUT='https://www.sugarcrm.com/au/request-demo/'

class TestMyTest(unittest.TestCase):

    def setUp(self):

        self.browser=webdriver.Firefox(executable\_path='geckodriver.exe')

    def test\_m1(self):

        self.browser.get(AUT)

        self.browser.maximize\_window()

        sleep(3)

        print(self.browser.title)

        button=self.browser.find\_element(By.ID,'CybotCookiebotDialogBodyLevelButtonLevelOptinAllowAll').click()

        sleep(3)

        self.browser.find\_element(By.ID,'doi0').click()

        print(self.browser.find\_element(By.ID,'doi0').is\_selected())

    def tearDown(self):

        self.browser.close()

        self.browser.quit()

if \_\_name\_\_ == "\_\_main\_\_":

    unittest.main()

**Question 2:**

Open the below AUT and click on radio button for Yes.

<https://demoqa.com/radio-button>

**Solution:**

AUT='https://demoqa.com/radio-button'

driver=webdriver.Firefox(executable\_path='geckodriver.exe')

driver.get(AUT)

print('\*\*\*\*\*\*Selecting Radio button\*\*\*\*\*\*\*\*')

driver.find\_element(By.XPATH,"//label[@class='custom-control-label'][1]").click()

sleep(3)

driver.close()

driver.quit()

#######################

# Drop down in selenium ##

#######################

In selenium for working with drop down it provides Select class.

It has below three methods for selecting the element from down -

class selenium.webdriver.support.select.Select(webelement)

select\_class\_variable=Select(web\_element\_locator)

**Methods for selecting element**

|  |  |
| --- | --- |
| select\_class\_instance.select\_by\_index(index) | Selecting by index, index>=0 |
| select\_class\_instance.select\_by\_value(value) | Selecting by value |
| select\_class\_instance.select\_by\_visible\_text(text) | Selecting by visible text |

Whenever are using these methods, we need to call these methods on the Select class type of variable.

**Methods for deselecting elements**

|  |  |
| --- | --- |
| deselect\_all()  deselect\_by\_index(index)  deselect\_by\_value(value)  deselect\_by\_visible\_text(text) |  |

**Question:**

From below url select the value from "Best Time to Contact" dropdown using selenium.

<https://fs2.formsite.com/meherpavan/form2/index.html?1537702596407>

<select id="RESULT\_RadioButton-9" name="RESULT\_RadioButton-9" class="drop\_down">

<option></option>

<option value="Radio-0">Morning</option>

<option value="Radio-1">Afternoon</option>

<option value="Radio-2">Evening</option>

</select>

**Solution:**

from selenium import webdriver

from selenium.webdriver.support.ui import Select

from selenium.webdriver.common.by import By

from time import sleep

import unittest

AUT='https://fs2.formsite.com/meherpavan/form2/index.html?1537702596407'

class TestMyTest(unittest.TestCase):

    def setUp(self):

        self.browser=webdriver.Firefox(executable\_path='geckodriver.exe')

    def test\_m1(self):

        self.browser.get(AUT)

        self.browser.maximize\_window()

        sleep(3)

        print(self.browser.title)

        #locating the drop down menu

        #creating selenium Select instance for drop down menu

        drop=Select(self.browser.find\_element(By.ID,'RESULT\_RadioButton-9'))

        #selecting by visiable test

        drop.select\_by\_visible\_text('Afternoon')

        sleep(3)

    def tearDown(self):

        self.browser.close()

        self.browser.quit()

if \_\_name\_\_ == "\_\_main\_\_":

    unittest.main()

########################

# Wait in Selenium #

########################

Sometimes in real world we need to wait when pages are loaded so that all the elements are loaded.

**Example**- If application based on AJAX then these scenario happens frequently b/c when page is loaded by browser then, elements within that may load at different time/speed.

If an element is not yet present in the DOM, a locate function will raise an ElementNotVisibleException exception. We can solve it by using wait.

In some cases element could be present in HTML document but not in position to interact in such cases "ElementNotInteractableException" is raised.

There could be any of below reasons for this:

1. The element is not visible / not displayed.
2. The element is off-screen.
3. The element is behind another element or hidden.
4. Some other action needs to be performed by the user first to enable it.

**Types of waits:**

There are two types of wait in selenium.

1. Implicit wait
2. Explicit wait

**Explicit wait:**

An explicit wait is a code you define to wait for a certain condition to occur before proceeding further in the code.

The extreme case of this is time.sleep(), which sets the condition to an exact time period to wait.

Sometimes it is unnecessary to wait the full extent of the default timeout, as the penalty for not hitting a successful condition can be expensive. We can take use of WebDriverWait class to wait explicitly.

WebDriverWait(driver, timeout=3).until(some\_condition)

------ Syntax

class selenium.webdriver.support.wait.WebDriverWait(driver, timeout: float, poll\_frequency: float = 0.5, ignored\_exceptions: Optional[Iterable[Type[Exception]]] = None)

WebDriverWait calls the ExpectedCondition every 500 milliseconds until it returns success. ExpectedCondition will return true (Boolean) in case of success or not null if it fails to locate an element

**Methods of Webdriver class:**

We have below method of webdriver class.

until(expected\_contion, message: str = '')

Calls the method provided with the driver as an argument until the return value does not evaluate to False.

until\_not(expected\_contion, message: str = '')

Calls the method provided with the driver as an argument until the return value evaluates to False.

**Expected condition:**

selenium.webdriver.support.expected\_conditions

There are some common conditions that are frequently of use when automating web browsers.

<https://www.selenium.dev/selenium/docs/api/py/webdriver_support/selenium.webdriver.support.expected_conditions.html?highlight=expecte>

The expected\_conditions module contains a set of predefined conditions to use with WebDriverWait.

Listed below are some common conditions that are frequently used when automating web browsers. Selenium Python binding provides some convenience class methods from expected\_condition class-

* presence\_of\_element\_located(locator)
* visibility\_of\_element\_located(locator)
* visibility\_of(locator)
* presence\_of\_all\_elements\_located(locator)
* text\_to\_be\_present\_in\_element
* text\_to\_be\_present\_in\_element\_value
* frame\_to\_be\_available\_and\_switch\_to\_it
* invisibility\_of\_element\_located
* element\_to\_be\_clickable
* staleness\_of
* element\_to\_be\_selected
* element\_located\_to\_be\_selected
* element\_selection\_state\_to\_be
* element\_located\_selection\_state\_to\_be
* alert\_is\_present()
* element\_exists()
* element\_is\_visible
* title\_contains
* title\_is
* element-staleness
* visible\_text
* title\_contains

**Example:**

driver.navigate("file:///race\_condition.html")

WebDriverWait(driver, timeout=10).until(document\_initialised)

el = driver.find\_element(By.TAG\_NAME, "p")

assert el.text == "Hello from JavaScript!"

Example 2:

AUT='https://www.geeksforgeeks.org/'

class TestMyTest(unittest.TestCase):

    def setUp(self):

        self.browser=webdriver.Firefox(executable\_path='geckodriver.exe')

    def test\_m1(self):

        self.browser.get(AUT)

        self.browser.maximize\_window()

        try:

            element = WebDriverWait(self.browser,10).

            until(EC.presence\_of\_element\_located((By.ID, "myDynamicElement")))

        except Exception as e:

            print(e)

    def tearDown(self):

        self.browser.close()

        self.browser.quit()

if \_\_name\_\_ == "\_\_main\_\_":

    unittest.main()

**Custom Wait Conditions**

You can also create custom wait conditions when none of the previous convenience methods fit your requirements. A custom wait condition can be created using a class with \_\_call\_\_ method which returns False when the condition doesn’t match.

class element\_has\_css\_class(object):

  """An expectation for checking that an element has a particular css class.

  locator - used to find the element

  returns the WebElement once it has the particular css class

  """

  def \_\_init\_\_(self, locator, css\_class):

    self.locator = locator

    self.css\_class = css\_class

  def \_\_call\_\_(self, driver):

    element = driver.find\_element(\*self.locator)   # Finding the referenced element

    if self.css\_class in element.get\_attribute("class"):

        return element

    else:

        return False

# Wait until an element with id='myNewInput' has class 'myCSSClass'

wait = WebDriverWait(driver, 10)

element = wait.until(element\_has\_css\_class((By.ID, 'myNewInput'), "myCSSClass"))

**Implicit wait:**

By implicitly waiting, it tells driver to wait for a certain duration when trying to find any element. This can be useful when certain elements on the webpage are not available immediately and need some time to load.

The default wait time for the webdriver is zero., meaning disabled. Once set, the implicit wait is set for the life of the WebDriver object.

driver.implicitly\_wait(n) , n= number of seconds to wait

**Note:**

Implicitly wait is applied globally (for all element ), which means it is always available for all the web elements throughout the browser instance.

**Fluent wait:**

FluentWait instance defines the maximum amount of time to wait for a condition, as well as the frequency with which to check the condition.

wait = WebDriverWait(driver, timeout=10, poll\_frequency=1, ignored\_exceptions=[ElementNotVisibleException, ElementNotSelectableException])

|  |  |  |
| --- | --- | --- |
| Explicit wait | Implicit wait | Fluent wait |
| Wait for certain condition to occur | Wait’s for certain amount of time |  |

################################################################################

# Exception in Selenium #

################################################################################

Exceptions in Selenium Python are the errors that occur when one of method fails or an unexpected event occurs.

All instances in Python must be instances of a class that derives from BaseException

We have below types of exception in selenium.





################################################################################

# Unit testing #

################################################################################

Unit testing is the practice of writing code to test the unit of code(function) and then run those tests in an automated fashion. It is used to check that code is working as expected.

Python have unittest inbuilt library for unit testing.

It supports test automation, sharing of setup and closure codes for tests, grouping of tests into test suites, and independence of the tests from the reporting framework.

To achieve this, unittest supports some important concepts in an object-oriented way:-

**test fixture**

A test fixture represents the preparation needed before the beginning of the one or more tests, and any associate closure activity. This may involve, for example, creating a browser instance and closing the browsing session.

**test case**

A test case is an individual unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class, "TestCase", which may be used to create new test cases.

**test suite**

A test suite is a collection of test cases, test suites, or both. It is used to group tests that should be executed together.

**test runner**

A test runner is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

In python unittest library we have unittest.TestCase class which we need to extend in our own class while creating test cases.

Example:

class WidgetTestCase(unittest.TestCase):

def setUp(self):

self.widget = Widget('The widget')

def test\_default\_widget\_size(self):

self.assertEqual(self.widget.size(), (50,50),

'incorrect default size')

Note:

1. All test class should start with Test word by this way it tells test runner that it represents tests. e.g-

Class TestEmployee(unittest.TestCase):

    pass

1. All the test method must start with test.

class TestEmployee(unittest.TestCase):

    def test\_employeeid(self.id):

        pass

TestCase instances provide three groups of methods:

1. used to run the test,
2. used by the test implementation to check conditions and report failures, and some inquiry methods allowing information about the test itself to be gathered.

**Note:**

The order of test case execution is determined by python it’s may not be in order of test declaration(determined by sorting the test method names with respect to the built-in ordering for strings).

|  |  |
| --- | --- |
| import unittest  class TestEmployee(unittest.TestCase):      def testemployeeid(self):          print('testing id')        def testsalary(self):          print('testing salary')        def testname(self):          print('testing length of name')    if \_\_name\_\_ == "\_\_main\_\_":      unittest.main() | Output:  PS D:\Learnings\Selenium> python test.py  testing id  .testing length of name  .testing salary  .  ----------------------------------------------------------------------  Ran 3 tests in 0.001s  OK  PS D:\Learnings\Selenium>  **Note:**  Oder of test case is not in order of declaration. |

**Methods in the first group (running the test) are:**

1. setUp()
2. tearDown()
3. setUpClass()
4. tearDownClass()
5. run(result=None)
6. skipTest(reason)
7. subTest(msg=None, \*\*params)
8. debug()

**setUp()**

Method called to prepare the test fixture. This is called immediately before calling the test method; other than AssertionError or SkipTest, any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

If there is any error in setup then the execution of all the test methods including tearDown() method will be ignored.

**tearDown()**

Method called immediately after the test method has been called and the result recorded.

This method will only be called if the setUp() succeeds, regardless of the outcome of the test method.

This is called even if the test method raised an exception. Any exception, other than AssertionError or SkipTest, raised by this method will be considered an additional error rather than a test failure (thus increasing the total number of reported errors). T. The default implementation does nothing.

|  |  |
| --- | --- |
| class TestMyTest(unittest.TestCase):      #set up method with a print statement      def setUp(self):          print("doing all setup")      def test\_one(self):          print("inside test one")      def test\_two(self):          print("inside test two")      def test\_three(self):          print("inside test three")        #tear down method with a print statement      def tearDown(self):          print("this is tear down method") | Output:  PS D:\Learnings\Selenium>  .doing all setup  inside test three  this is tear down method  .doing all setup  inside test two  this is tear down method  .  ----------------------------------------------------------------------  Ran 3 tests in 0.002s  OK  **Note:**  setup is called first and teardown is called immediately after all test cases are executed. |

**setUpClass()**

A class method called before all tests in an individual class are run.

setUpClass is called with the class as the only argument and must be decorated as a classmethod:

@classmethod

def setUpClass(cls):

    ...

**tearDownClass()**

A class method called after tests in an individual class have run. tearDownClass is called with the class as the only argument and must be decorated as a classmethod:

@classmethod

def tearDownClass(cls):

    ...

|  |  |
| --- | --- |
| class TestMyTest(unittest.TestCase):      #set up method with a print statement      @classmethod      def setUpClass(cls):          print('this is septup class method')      def setUp(self):          print("doing all setup")      def test\_one(self):          print("inside test one")      def test\_two(self):          print("inside test two")      def test\_three(self):          print("inside test three")      @classmethod      def tearDownClass(cls):          print('this is teardownclass method') | PS D:\Learnings\Selenium> python test.py  this is septup class method  doing all setup  inside test one  this is tear down method  .doing all setup  inside test three  this is tear down method  .doing all setup  inside test two  this is tear down method  .this is teardownclass method |

**Order of execution:**

setupclass() ----- setup() ------ test cases ----- teardown() ------ teardownclass()

**setUpModule()**

A method called before the execution of any component of a module.

**tearDownModule()**

A method called after the execution of all the components of a module.

**Order of execution:**

setupmodule() ---setupclass() ----- setup() ------ test cases ----- teardown() ------ teardownclass() --- teardowndomule()

def setUpModule():

    print("In setUpModule")

def tearDownModule():

    print("In tearDownModule")

class TestMyTest(unittest.TestCase):

    #set up method with a print statement

    @classmethod

    def setUpClass(cls):

        print('this is septup class method')

    def setUp(self):

        print("doing all setup")

    def test\_one(self):

        print("inside test one")

    def test\_two(self):

        print("inside test two")

    def test\_three(self):

        print("inside test three")

    @classmethod

    def tearDownClass(cls):

        print('this is teardownclass method')

    #tear down method with a print statement

    def tearDown(self):

        print("this is tear down method")

if \_\_name\_\_ == "\_\_main\_\_":

    unittest.main()

Note:

1. If an exception arises during a setUpModule then the tests in the class are not run and the tearDownModule method will not run.
2. If an exception arises during a setUpClass then the tests in the class are not run and the tearDownClass method will not run.

**Question:**

Suppose developer have written code to calculate the area of rectangle in rect.py file and we have to test it. Write down developer code and test it.

rect.py -------- Developer code

from math import  pi

class Rectange:

    #this method will calculate the aera of rectangle and return

    def rect\_area(self,a,b):

        return a\*b

**Solution:**

TestVerify.py ------ tester code to test

import unittest

from rect.py import Rectangle

class Test\_Rectange(unittest.TestCase):

    def testArea(self):

        #test area when radius >=0

        cObject=Rectangle()

        self.assertAlmostEqual(cObject.rect\_area(1,1), 1)

        print('done')

if \_\_name\_\_ == "\_\_main\_\_":

    unittest.main()

**skipTest(reason)**

Calling this during a test method or setUp() skips the current test. See Skipping tests and expected failures for more information.

**run(result=None)**

Run the test, collecting the result into the TestResult object passed as result. If result is omitted or None, a temporary result object is created (by calling the defaultTestResult() method) and used. The result object is returned to run()’s caller.

**Other reporting methods:**

The TestCase class provides several assert methods to check for and report failures. The following table lists the most commonly used methods (see the tables below for more assert methods):

| **Method** | **Checks that** | **New in** |
| --- | --- | --- |
| [assertEqual(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertEqual) | a == b |  |
| [assertNotEqual(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertNotEqual) | a != b |  |
| [assertTrue(x)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertTrue) | bool(x) is True |  |
| [assertFalse(x)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertFalse) | bool(x) is False |  |
| [assertIs(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertIs) | a is b | 3.1 |
| [assertIsNot(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertIsNot) | a is not b | 3.1 |
| [assertIsNone(x)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertIsNone) | x is None | 3.1 |
| [assertIsNotNone(x)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertIsNotNone) | x is not None | 3.1 |
| [assertIn(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertIn) | a in b | 3.1 |
| [assertNotIn(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertNotIn) | a not in b | 3.1 |
| [assertIsInstance(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertIsInstance) | isinstance(a, b) | 3.2 |
| [assertNotIsInstance(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertNotIsInstance) | not isinstance(a, b) | 3.2 |

**Assertion in Selenium**

In selenium we have two type of assertion-

* Hard Assertion
* Soft Assertion

**Hard Assertion:**

Hard Assertion is an Assertion that throws the AssertException when the assert statement fails.

If not handled then it will stop the program execution below that line.

Example:

import unittest

class Test\_class(unittest.TestCase):

    def setUp(self):

        pass

    def test\_m1(self):

        print('this is test\_m1 function')

        self.assertEqual(1, 2)#this wull throw assertion exception

        print('assertion passed')

if \_\_name\_\_ == "\_\_main\_\_":

    unittest.main()

Here assertion will fail (assertEqual(1,2)) and will stop the program execution below that line.

We can handle it in below way using AssertionError exception

import unittest

class Test\_class(unittest.TestCase):

    def setUp(self):

        pass

    def test\_m1(self):

        print('this is test\_m1 function')

        try:

            self.assertEqual(1, 2)#this wull throw assertion exception

        except **AssertionError** as e:

            print(e)

        print('risky code executed')

if \_\_name\_\_ == "\_\_main\_\_":

    unittest.main()

**Soft Assertion**

When we want to continue the program execution though there is assertion error then we should use soft assertion.

We need to use TestCase class from softcase library while inheriting in our class.

Example:

import softest

class Test\_class(softest.TestCase):

    def setUp(self):

        pass

    def test\_m1(self):

        print('this is test\_m1 function')

        self.soft\_assert(self.assertEqual,1,2)#this wull throw assertion exception

        print('risky code executed')

if \_\_name\_\_ == "\_\_main\_\_":

    softest.main()

Now even if assertion fails it will execute all test cases and all line.

**#################################**

**# Skipping tests and expected failures #**

**#################################**

Unittest supports skipping individual test methods and even whole classes of tests. In addition, it supports marking a test as an “expected failure,”.

Test which is failed on some condition or forcibly will not be considered as failure in result.

**Why need to skip any test:**

Any groundwork or condition that has not been set up properly will lead to a failure of all the subsequent test methods, to avoid that we can skip the test case or test suite.

The following decorators and exception implement test skipping and expected failures:

@unittest.skip(reason)

Unconditionally skip the decorated test. reason should describe why the test is being skipped.

@unittest.skipIf(condition, reason)

Skip the decorated test if condition is true.

@unittest.skipUnless(condition, reason)

Skip the decorated test unless condition is true.

@unittest.expectedFailure

Whenever a function is expected to raise an error, the same can be described in the unit testing using a decorator @unittest.expectedFailure.

A test method marked with this decorator will be expected to raise/receive an error only then the status of the test method will be a success else the status will be a failure.

Example-

Let us assume below code

from math import  pi

import unittest

class circle:

    def circle\_area(self,radius):

        if radius < 0:

            #throws an error , if radius is a negative value

            raise ValueError("The radius cannot be negative.")

        return pi \* (radius \*\* 2)

class test\_Circle(unittest.TestCase):

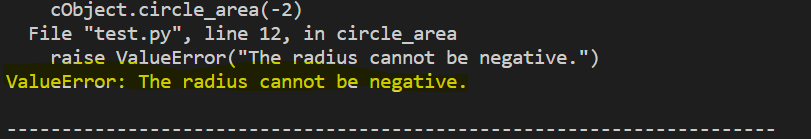
    def testArea(self):

        #test area when radius >=0

        cObject=circle()

        self.assertAlmostEqual(cObject.circle\_area(1), pi)

When we the we supply radius -ve value then test will be marked as failed.



We can handle this expected failure by use of @unittest.expectedFailure decorator as-

from math import  pi

class circle:

    def circle\_area(self,radius):

        if radius < 0:

            #throws an error , if radius is a negative value

            raise ValueError("The radius cannot be negative.")

        return pi \* (radius \*\* 2)

class test\_Circle(unittest.TestCase):

    @unittest.expectedFailure

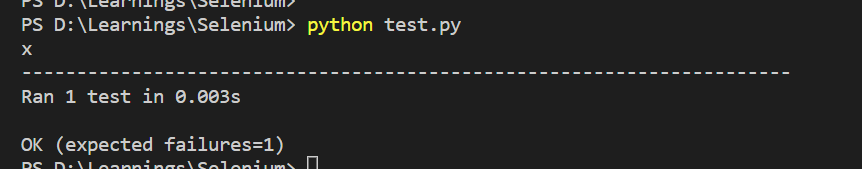
    def testArea(self):

        #test area when radius <=0

        cObject=circle()

        self.assertAlmostEqual(cObject.circle\_area(-1), pi)

Now if we run the test then it will not be marked as failed but will be marked as expectedFailure.



Example 2:

When we run the below code with +ve value of radius then it will be marked as failure.

Reason for failure is in definition of @unittest.expectedFailure.

A test method marked with this decorator will be expected to raise/receive an error only then the status of the test method will be a success else the status will be a failure.

class circle:

    def circle\_area(self,radius):

        if radius < 0:

            #throws an error , if radius is a negative value

            raise ValueError("The radius cannot be negative.")

        return pi \* (radius \*\* 2)

class test\_Circle(unittest.TestCase):

    @unittest.expectedFailure

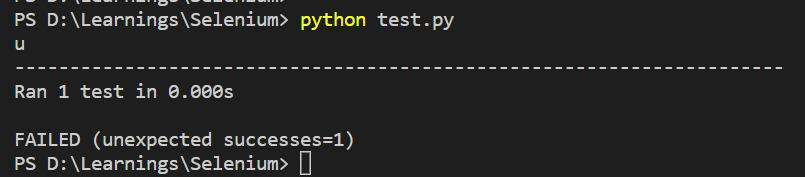
    def testArea(self):

        #test area when radius >=0

        cObject=circle()

        self.assertAlmostEqual(cObject.circle\_area(1), pi)

Output:



exception unittest.SkipTest(reason)

This exception is raised to skip a test.

**Question**

For below circle class and circle\_area method. Create a test method which will skip when radius is less then 0.

class circle:

    def circle\_area(self,radius):

        if radius < 0:

            #throws an error , if radius is a negative value

            raise ValueError("The radius cannot be negative.")

        return pi \* (radius \*\* 2)

Solution:

class circle:

    def circle\_area(self,radius):

        if radius < 0:

            #throws an error , if radius is a negative value

            raise ValueError("The radius cannot be negative.")

        return pi \* (radius \*\* 2)

class test\_Circle(unittest.TestCase):

    circleObject=None

    radius=-2 #setting radius less than 0

    @classmethod

    def setUpClass(cls):

        cls.circleObject=circle()

    @unittest.skipIf(radius<0,"negative value")

    def test\_Value(self):

        circle().circle\_area(-2)

    @classmethod

    def tearDownClass(cls):

        cls.circleObject=None

#############################################################################

# HTML Tables #

#############################################################################

Tables are used to represent the data in tabular format. We can create it using below tags-

1. tr
2. th
3. td
4. table
5. tbody

We can classify table of two types-

1. Static table --- table row and column data are fixed
2. Dynamic table --- table’s row and data are not fixed, dynamic in nature.

**Questions:**

Get the table header and row data.

from selenium import webdriver

browser=webdriver.Chrome(executable\_path='chromedriver.exe')

browser.get('link\_for\_table')

print(browser.title)

#get the table header

cols=browser.find\_elements\_by\_tag\_name('th')

for j in cols:

    print(j.text)#get the table head data for each table (in case there are many table herder in link)

#get the table row

rows= browser.find\_elements\_by\_tag\_name('tr')

for each in rows:

    print(each.text)

browser.close()

browser.quit()

**Iterating table elements:**

If table is nested with tags then we can call methods by nesting, if it logically fits

Let say we have below table

<table id='customers'>

    <tbody><tr>

      <th>Company</th>

      <th>Contact</th>

      <th>Country</th>

    </tr>

    <tr>

      <td>Alfreds Futterkiste</td>

      <td>Maria Anders</td>

      <td>Germany</td>

    </tr>

    <tr>

      <td>Centro comercial Moctezuma</td>

      <td>Francisco Chang</td>

      <td>Mexico</td>

    </tr>

    <tr>

      <td>Ernst Handel</td>

      <td>Roland Mendel</td>

      <td>Austria</td>

    </tr>

    <tr>

      <td>Island Trading</td>

      <td>Helen Bennett</td>

      <td>UK</td>

    </tr>

    <tr>

      <td>Laughing Bacchus Winecellars</td>

      <td>Yoshi Tannamuri</td>

      <td>Canada</td>

    </tr>

    <tr>

      <td>Magazzini Alimentari Riuniti</td>

      <td>Giovanni Rovelli</td>

      <td>Italy</td>

    </tr>

  </tbody>

  </table>

For above web elements we can call in below way-

body=self.driver.find\_element(By.XPATH,"//\*[@id='customers']/tbody")

#now on body we can call locators

body.find\_element(By.TAG\_NAME,'tr[1]')

#OR#

row=self.driver.find\_element(By.XPATH,"//\*[@id='customers']/tbody/tr[1]")

#now on body we can call locators

row.find\_elements(By.TAG\_NAME,'td')

**Method 2:**

Get the number of rows then get the data one by one for each row

def test\_m1(self):

        sleep(1)

        print('headers are: ')

        headers=self.driver.find\_elements(By.XPATH,"//\*[@id='customers']/tbody/tr[1]/th")

        for each in headers:

            print(each.text,end=' '\*25)

        rows=self.driver.find\_elements(By.XPATH,"//\*[@id='customers']/tbody/tr")

        print('number of rows are: ',len(rows))

        for each in range(2,len(rows)+1):

            print(each)

            path="//\*[@id='customers']/tbody/tr["+str(each)+']/td'

            datas=self.driver.find\_elements(By.XPATH,path)

            for i in datas:

                print(i.text,end=' '\*3)

            print()

################################################################################# Working with windows and tabs #

################################################################################

WebDriver does not make the distinction between windows and tabs.

If site opens a new tab or window, then we need to switch control to new window, by default it remains with parent window.

Selenium will let you work with it using a window handle. Each window has a unique identifier which remains persistent in a single session

Selenium allows to switch the control b/c windows using below methods-

driver.switch\_to.window("new\_window") --- ------ Switch to new window handle

driver.window\_handles ------ get all window handle

driver. current\_window\_handle ------ get the current window handle

**Questions:**

Open the below link and click on Forum button and get the title of new window.

<https://www.freecodecamp.org/news/how-to-use-html-to-open-link-in-new-tab/>

Solution:

AUT='https://www.freecodecamp.org/news/how-to-use-html-to-open-link-in-new-tab/'

class TestMyTest(unittest.TestCase):

    def setUp(self):

        self.browser=webdriver.Firefox(executable\_path='geckodriver.exe')

    def test\_m1(self):

        self.browser.get(AUT)

        sleep(3)

        #get he curent winod handle

        parent\_window\_handle=self.browser.current\_window\_handle

        self.browser.find\_element(By.ID,'nav-forum').click()

        sleep(2)

        #now get all window handler in current session

        all\_window\_handle=self.browser.window\_handles

        for each in all\_window\_handle:

            #go to child window handle

            if each!=parent\_window\_handle:

                #switch to child window

                self.browser.switch\_to.window(each)

                print(self.browser.title)

                sleep(2)

                #close the child window

                self.browser.close()

                #switch back to parent window handle

                self.browser.switch\_to.window(parent\_window\_handle)

                sleep(2)

    def tearDown(self):

        #self.browser.close()

        self.browser.quit()

if \_\_name\_\_ == "\_\_main\_\_":

    unittest.main()

**Create new window (or) new tab and switch**

By using below methods, we can create a new window (or) tab and will focus the new window or tab on screen. You don’t need to switch to work with the new window (or) tab.

# Opens a new tab and switches to new tab

driver.switch\_to.new\_window('tab')

# Opens a new window and switches to new window

driver.switch\_to.new\_window('window')

If you have more than two windows (or) tabs opened other than the new window, you can loop over both windows or tabs that WebDriver can see, and switch to the one which is not the original.

**#####################**

**# Window management #**

**#####################**

Window management is related with getting the size of window, change the size of window etc.

**Get window size**

# Access each dimension individually

width = driver.get\_window\_size().get("width")

height = driver.get\_window\_size().get("height")

**OR**

# Or store the dimensions and query them later

size = driver.get\_window\_size()

width1 = size.get("width")

height1 = size.get("height")

**Set window size**

driver.set\_window\_size(1024, 768)

**Get window position**

# Access each dimension individually

x = driver.get\_window\_position().get('x')

y = driver.get\_window\_position().get('y')

# Or store the dimensions and query them later

position = driver.get\_window\_position()

x1 = position.get('x')

y1 = position.get('y')

**Other methods:**

driver.minimize\_window()

driver.fullscreen\_window()

driver.save\_screenshot('image\_path\_to\_save')

##############################################################################

# Alert Handling #

##############################################################################

Alert on webpage is used to get attention or getting some operator/user input.

When alert is popped up it prevents driver to perform the next action until it is handled. If we try to do anything without handling then it gives “selenium.common.exceptions.UnexpectedAlertPresentException.” exceotion.

Alert can occur on webpage-

1. On page load
2. On page close
3. On click of an element
4. On right-click (when right-clicking is disabled)
5. On wrong entry of a field
6. On saving of information

**Types of alerts:**

Alert can be of below types-

1. Simple alert
2. Confirmation dialog
3. Prompt

**Properties of Alerts**

1. We cannot identify alerts using inspect tools
2. We cannot write xpaths for alerts
3. It is not a window
4. We cannot handle alerts using javaScript Executor.
5. We cannot consider it as an alert if any of the above properties mismatches.

**Handling alert in selenium:**

We can handle alert in selenium using below method. It gives the alert handle, using alert handle we can accept, dismiss etc.

alert\_handle = driver.switch\_to\_alert() --- To get the alert handler

alert\_handle.accept() --- To accept the alert

alert\_handle.dismiss() ---- dismiss the alert

alert\_handle.text ------- get the alert text

alert\_handle.send\_keys("<<text>>" ---- enter some data into alert box

Example:

# Click the link to activate the alert

driver.find\_element(By.LINK\_TEXT, "See an example alert").click()

# Wait for the alert to be displayed and store it in a variable

alert = wait.until(expected\_conditions.alert\_is\_present())

# Store the alert text in a variable

text = alert.text

# Press the OK button

alert.accept()

**#############################################################################**

**# Handling Frame #**

**#############################################################################**

HTML Frames are used to divide the web browser window into multiple sections where each section can be loaded separately.

A document with frames has a HEAD, a FRAMESET, and an optional BODY.

we can handle the frames by switch\_to command in selenium.

**Handling of frame:**

Using selenium we can handle frame in 3 different ways-

1. By using the Index of Frame ---- driver.switch\_to\_frame("index\_of\_frame")
2. By using the Id of Frame ------ driver.switch\_to\_frame("id\_of\_frame")
3. By using the name of Frame --- driver.switch\_to\_frame("frame\_name")
4. By using tag name ------ driver.switch\_to\_frame("tag\_name")

driver.switch\_to\_default\_content() ----- For switching to parent frame

**Selecting frame using name or ID**

# Switch frame by id

driver.switch\_to.frame('buttonframe')

# Now, Click on the button

driver.find\_element(By.TAG\_NAME, 'button').click()

**Using an index**

switching to second iframe based on index

iframe = driver.find\_elements(By.TAG\_NAME,'iframe')[1]

# switch to selected iframe

driver.switch\_to.frame(iframe)

**###############################################################################**

**# Mouse hovering in Selenium #**

**###############################################################################**

In some cases, webpages are designed to trigger an event that shows some more option. Most common cases are dropdown menu.

If we hover on “Drop down link “ on below link then we can see it.

<https://www.w3schools.com/cssref/tryit.asp?filename=trycss_sel_hover_dropdown>

**Handling mouse hover**

* Hover the mouse on specified button/tab to get all item list.
* Click on the item list.

**Action chain**

ActionChains is a class to automate low-level interactions such as mouse movements, mouse button actions, keypress, and context menu interactions.

When you call methods for actions on the ActionChains object, the actions are stored in a queue in the ActionChains object. When you call perform(), the events are fired in the order in which they are queued up.

class selenium.webdriver.common.action\_chains.ActionChains(driver, duration=250)

Example:

actions.move\_to\_element(firstWebElement).move\_to\_element(secondWebElemnet).click(ThirdWebElement).perform()

**Question**

Suppose there is on website, we have to implement the hover action and then click on second element of list item.

**Solution:**

actions = ActionChains(self.driver) -------- Action handle

actions.move\_to\_element(web\_element\_locator) ------ moving to first element

actions.click(web\_element\_locator) ----- click on second element

actions.perform() ---- move the mouse over that item, it will show hover action

**Below have below all methods available in Action Chain class.**

|  |  |
| --- | --- |
|  |  |

################################################################################

# Drag and Drop action in selenium #

################################################################################

Drag and drop is a very common feature. It is when you "grab" an object in one location and drag it to a different location.

Drag and drop can be achieved by Action\_chains in Selenium.

drag\_and\_drop(source, target) ------ for drag and drop operation

Example:

mouse = ActionChains(self.driver).drag\_and\_drop(source, target)

mouse.perform()

**###############################################################################**

**# Working with Excel #**

**###############################################################################**

Python have many packages to work with excel sheet.

xlrd ---- for retrieving data and formatting the data in Excel files(.xls or .xlsx)

xlwt --- for writing the data and formatting the data in Excel files(.xls or .xlsx

To Create Workbook:

WorkBook = xlwt.Workbook();

To Add New Sheet:

WorkSheet = WorkBook.add\_sheet("<<Sheet Name>>", Cell\_overwrite\_ok)

To Write in Sheet:

WorkSheet.write(row\_index,column\_index,"<<value>>")

Minimum value of row\_index,column\_index start with 0

To Save the Workbook:

WorkBook.save("Excel\_Workbook.xls"/"Excel\_Workbook.xlsx")

--This will help save the workbook as ".xls" or ".xlsx"

**#To Write in Existing New Workbook#**

To Copy Workbook:

WorkBook = xlrd.open\_workbook(filepath)

writebook = copy(WorkBook)

---Copy method will help to copy the opened xlrd objects to xlwt.Workbook objects

sheet=writebook.add\_sheet(sheet\_name) ---- to add a new sheet

sheet=writebook.get\_sheet() ---- to write in existing sheet.

sheet.write(row,col,value) ------ to write in sheet

writebook.save(file\_path) --- After completing write operation, save it.

To Fetch the Sheet:

worksheet = writebook.get\_sheet(<<sheet index>>)

To Open Workbook:

workbook = xlrd.open\_workbook("filename.xls");

or

workbook = xlrd.open\_workbook("/path/filename.xls");

To Find WorkSheet:

sheet = workbook.sheet\_by\_name("<<name of the sheet>>")

or

sheet = workbook.sheet\_by\_index(<<index number of the sheet>>)

To Find Number of Sheets and Sheet Names:

workbook.nsheets

and

workbook.sheet\_names()

---nsheets: Get the number of sheets in workbook

---sheet\_names(): Fetch all the sheet names in the workbook in the form of array.

To get row and column count

rowcount = sheet.nrows

colcount = sheet.ncols

To Fetch the Data from Cell:

value = sheet.cell\_value(row\_number,column\_number)

**To open workbook**

workbook = xlrd.open\_workbook("filename.xls") -- to open a workbook

**To find the worksheet**

sheet = workbook.sheet\_by\_name("<<name of the sheet>>")

or

sheet = workbook.sheet\_by\_index(<<index number of the sheet>>)

**To find number of sheet and sheet names**

workbook.nsheets ----- to find number of sheet

and

workbook.sheet\_names() ---- to find sheet names

**To find number of rows**

rowcount = sheet.nrows --- to get number of rows

colcount = sheet.ncols ----- to get number of columns

**To fetch data from cell**

value = sheet.cell\_value(row\_number,column\_number)

**To Create Workbook**

WorkBook = xlwt.Workbook();

**To add new sheet**

WorkSheet = WorkBook.add\_sheet("<<Sheet Name>>", Cell\_overwrite\_ok)

**To write in sheet**

WorkSheet.write(row\_index,column\_index,"<<value>>")

**To save workbook**

WorkBook.save("Excel\_Workbook.xls"/"Excel\_Workbook.xlsx")

**To Copy workbook**

WorkBook = xlrd.open\_workbook(filepath)

writebook = copy(WorkBook)

**To Fetch sheet**

worksheet = writebook.get\_sheet(<<sheet index>>)

################################################################################# Listeners #

################################################################################

Listener are the functions or block of code which capture/listens the event/result when it occurs.

Listeners can capture the webdriver and unit test result actions.

* Listeners for webdriver actions are handled by **Event Listeners**.
* Listeners for test results are handled by **Unit TestResult Listeners**.

Although event listener captures the event but doesn’t provide any action by itself, action must be provided by user.

**Event listeners:**

Event listeners are functions in selenium python bindings that wait for an event to occur; that event may be a user clicking or finding, changing the value, or an Exception. The listeners are programmed to react to an input or signal.

There is a function to react when an even is about to happen; also, there is a function to react after the event.

AbstractEventListener is inbuilt class which provides pre and the post event listeners.

To use these listeners, we have to implement those listeners by inheriting AbstractEventListener.

**EventFiringWebDriver:**

EventFiringWebDriver is a wrapper on the webdriver which provides all the methods that the webdriver provides. EventFiringWebDriver provides two more functions to register and unregister the listeners implementing the class.

**Methods of AbstractEventListener**

We have below methods in this class which we can use-

def before\_navigate\_to(self, url, driver):

pass

def after\_navigate\_to(self, url, driver):

pass

def before\_navigate\_back(self, driver):

pass

def after\_navigate\_back(self, driver):

pass

def before\_navigate\_forward(self, driver):

pass

def after\_navigate\_forward(self, driver):

pass

def before\_find(self, by, value, driver):

pass

def after\_find(self, by, value, driver):

pass

def before\_click(self, element, driver):

pass

def after\_click(self, element, driver):

pass

def before\_change\_value\_of(self, element, driver):

pass

def after\_change\_value\_of(self, element, driver):

pass

def before\_execute\_script(self, script, driver):

pass

def after\_execute\_script(self, script, driver):

pass

def before\_close(self, driver):

pass

def after\_close(self, driver):

pass

def before\_quit(self, driver):

pass

def after\_quit(self, driver):

pass

def on\_exception(self, exception, driver):

pass

**How to use AbstractEventListener**

Inherit AbstractEventListener class in your own class and we can use any methods of AbstractEventListener class.

from selenium.webdriver.support.abstract\_event\_listener import AbstractEventListener

class MyListener(AbstractEventListener):

    pass

    #use any methods of AbstractEventListener class here as instance method

    def before\_navigate\_to(self, url, driver):

        print("before going to url", url)

    def after\_navigate\_to(self, url, driver):

        print("after going to url ", url)

**Test Listeners:**

Test Listeners are also the event listeners, listener methods will be invoked when there are events happening around the tests. Events could be failure, pass, skip and so on.

**Methods for Test Listener:**

In unittest.TestCase class there are some listener, mention below-

class Test(unittest.TestCase):

def startTest(self, test):

pass

def stopTest(self, test):

pass

def addFailure(self, test, err):

pass

def addSuccess(self, test):

pass

def addSkip(self, test, reason):

pass

def addExpectedFailure(self, test, err):

pass

def addUnexpectedSuccess(self, test):

pass

################################################################################# HtmlTestRunner #

################################################################################

It generates a HTML report to show the result at a glance.

The simplest way to use it is to invoke with main method.

unittest.main(testRunner=HTMLTestRunner\_obj)

Note:

We can not change testRunner by any other string. It’s a kind of keyword.

from HtmlTestRunner import HTMLTestRunner

if \_\_name\_\_ == '\_\_main\_\_':

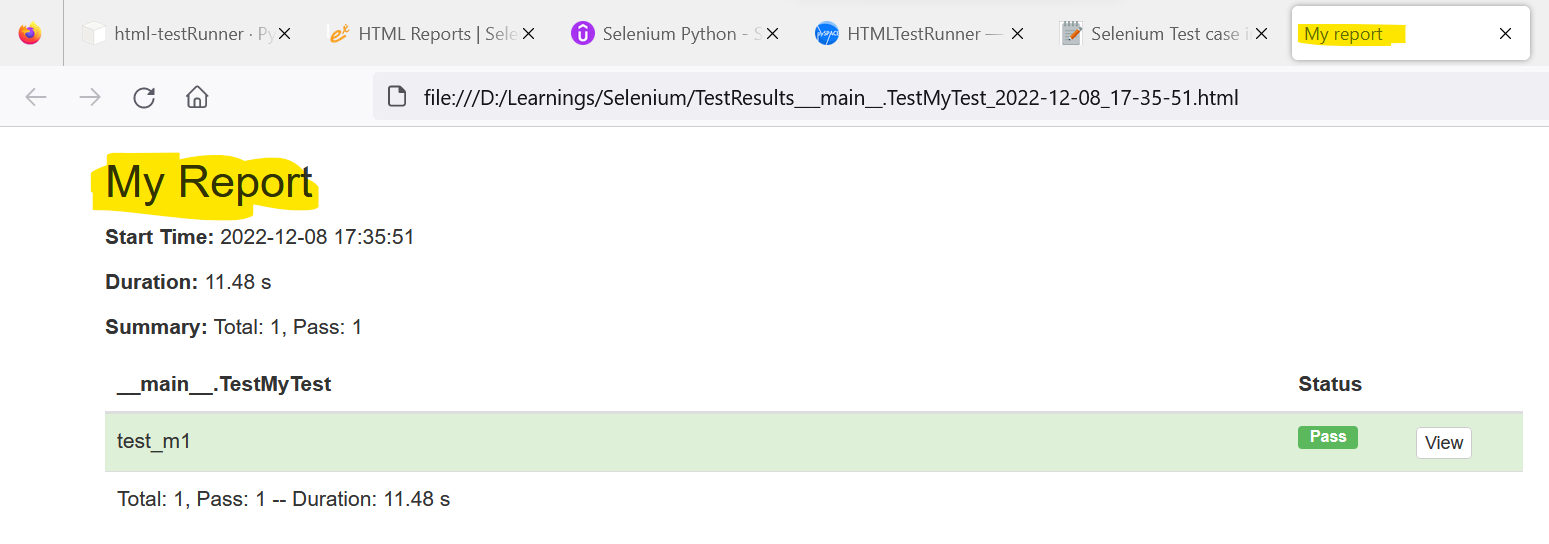
    unittest.main(testRunner=HTMLTestRunner(output='D:\\Learnings\\Selenium'+'/reports/',report\_title='My report',descriptions='My description'))

output – Path where report need to be generated. (Mandatory)

report\_title—title for generated report (Optional)

description --- description (optional)

testRunner --- it’s a keyword, don’t use any other string. It is should be object of HTMLTestRunner.



###########

# TestSuite #

###########

A test suite is a collection of test cases, test suites, or both. It is used to group tests that should be executed together. For creating test suite we will be using below two classes and their method-

* TestLoader ---- Used to create test suite from class and module
* TestRunner --- Used for aggregating test suite and run them

**TestLoader**

The TestLoader class is used to create test suites from classes and modules.

Basically ,this calls is used to load load from class and module.

We have below methods of this class to create test suite-

1. **TestLoader().loadTestsFromTestCase(MyTestCaseClass)**

Return a suite of all test cases contained in the TestCase-derived MyTestCaseClass.

A test case instance is created for each method named by getTestCaseNames()

1. **TestLoader().loadTestsFromModule(module, pattern=None)**

Return a suite of all test cases contained in the given module.

This method searches module for classes derived from TestCase and creates an instance of the class for each test method defined for the class.

**TestSuite**

This class represents an aggregation of individual test cases or loaded from TestLoader and test suites. The class presents the interface needed by the test runner to allow it to be run as any other test case.

Suit\_name=TestSuite((testcases\_in\_tuple)

**Methods of TestSuite class**

1. **addTest(test)**

Add a TestCase or TestSuite to the the suite

1. **addTests(tests)**

Add all the tests from an iterable of TestCase and TestSuite instances to this test suite.

1. **run(result)**

Run the tests associated with this suite, collecting the result into the test result object passed as result.

**class TextTestRunner**

A basic test runner implementation that outputs results to a stream. If stream is None, the default, sys.stderr is used as the output stream. It is usually used to run the test suite.

**Method of TextTestRunner**

1. **run(suit\_name)**

This is used to runt the test suite.

Example: Creating a test

Login.py

import unittest

class TestLogin(unittest.TestCase):

    def test\_login\_by\_email(self):

        print('this is login by email')

        self.assertTrue(1)

    def test\_login\_by\_fb(self):

        print('this is login by fb')

        self.assertTrue(1)

if \_\_name\_\_=="\_\_main\_\_":

    unittest.main()

Signup.py

import unittest

class TestSignUp(unittest.TestCase):

    def test\_singup\_by\_email(self):

        print('this is sign by email')

        self.assertTrue(1)

    def test\_signup\_by\_fb(self):

        print('this is sign by fb')

        self.assertTrue(1)

if \_\_name\_\_=="\_\_main\_\_":

    unittest.main()

mysuite.py ---- Creating suite for login.py and signup.py

from login import TestLogin

from signup import TestSignUp

import unittest

#Load the test cases

tc1=unittest.TestLoader().loadTestsFromTestCase(TestLogin)

tc2=unittest.TestLoader().loadTestsFromTestCase(TestSignUp)

#create a suite

s1=unittest.TestSuite((tc2,tc1))

s2=unittest.TestSuite((tc1))

#Now run the test suite

unittest.TextTestRunner().run(s1)

unittest.TextTestRunner().run(s2)

**Note:**

If two test suites are containing same test cases then run() will throw error.

s1=unittest.TestSuite((tc2,tc1))

s2=unittest.TestSuite((tc1,ct2))

#s1 and s2 containing same test cases

#Now run the test suite

unittest.TextTestRunner().run(s1) # This will give error

unittest.TextTestRunner().run(s2) # This will give error

below will run fine b/c both the test suite are not duplicate

s1=unittest.TestSuite((tc2,tc1))

s2=unittest.TestSuite((tc1))

#s1 and s2 are not containing same test cases

#Now run the test suite

unittest.TextTestRunner().run(s1)

unittest.TextTestRunner().run(s2)

################################################################################

# Working with Auto Suggestion #

################################################################################

Auto Suggestion is suggestion given by website when you try to search something anywhere on search box.

Example ---- Try to search anything on google. We will get lot of suggestions.

We can work with suggestion in two ways-

1. Manually hardcoding the value that want to seach
2. Selection from suggestions

**Manually hardcoding**

This is way of searching data what we want to search. This way doesn’t look at the suggestions.

**Steps:** -

* Locate the search box
* Pass the value want to search
* Press enter using selenium

#Locate the search box

seach=self.driver.find\_element(By.XPATH,"xpath]")

#enter value want to seach

seach.send\_keys("enter searching value")

#hit enter using selenium

seach.send\_keys(Keys.ENTER)

Example: -

class TestGoogle(unittest.TestCase):

    AUT="https://www.google.com/"

    def setUp(self):

        self.driver=webdriver.Firefox(executable\_path="geckodriver.exe")

        self.driver.get(TestYatra.AUT)

        self.driver.maximize\_window()

        self.driver.implicitly\_wait(3)

    def test\_check\_ticket(self):

        sleep(1)

        try:

            #Locate the search

            seach=self.driver.find\_element(By.XPATH,"//\*/input[@class='gLFyf']")

            #seach for moscow

            seach.send\_keys("Moscow")

            seach.send\_keys(Keys.ENTER)

            sleep(3)

        except NoSuchFrameException as e:

**Searching a value from suggestion box**

This is approach we type few character in search box and then select a value from suggestion box to search.

**Steps: -**

1. Locate the search box.
2. Enter few characters in search box from searching string.
3. Get all suggestions value in a list.
4. Pick one value from list what want to search and click on that web element.

#Locate the search

seach=self.driver.find\_element(By.XPATH,"//\*/input[@class='gLFyf']")

#seach for moscow

seach.send\_keys("Moscow")

#find all suggested value

suggestions=self.driver.find\_elements(By.XPATH,"//\*/ul[@class='G43f7e']/li")

#from list of web element iterate to check the value which want to seach

for each in suggestions:

if condition:

#click on that web lement

each.click()

break

Example:

def test\_check\_ticket(self):

        sleep(1)

        try:

            #Locate the search

            seach=self.driver.find\_element(By.XPATH,"//\*/input[@class='gLFyf']")

            #seach for moscow

            seach.send\_keys("Moscow")

            #find all suggested value

            suggestions=self.driver.find\_elements(By.XPATH,"//\*/ul[@class='G43f7e']/li")

            #from list of web elements check the text want to seach

            for each in suggestions:

#if suggestion string contain mule then search that

                if "mule" in each.text.lower():

                    #click on that web lement

                    each.click()

                    break

        except NoSuchFrameException as e:

            print(e)

**Note:**

If we are handling more than one auto suggestion then instead of directly performing click operation on web element we should click using help of ActionChain class.

If not done by using actionchain then we could get error b/c first autosuggestion field will obscure another autosuggestion field.

Question: Select the source and destination location from goibibo application where-

Source city --- should contain ‘new york’ in it

Destination city --- should contain ‘India’ in it.

Solution

#Select the source city/airport

        self.driver.find\_element(By.XPATH,"//\*/div[@class='sc-bYoBSM jmHUba']/div").click()

        action=ActionChains(self.driver)

        source=self.driver.find\_element(By.XPATH,"//\*/div[@class='sc-jJoQJp echhfS']/input")

        source.send\_keys('New')

        sources=self.driver.find\_elements(By.XPATH,"//\*/ul[@id='autoSuggest-list']/li")

        for each in sources:

            if 'new york' in each.text.lower():

#perform the click operation using action class so that it don’t obscure

                action.move\_to\_element(each).click().perform()

                break

        sleep(3)

        #Now select the destination city/airport

        self.driver.find\_element(By.XPATH,"//\*/div[@class='sc-bYoBSM elpDYd']").click()

        dest=self.driver.find\_element(By.XPATH,"//\*/div[@class='sc-jJoQJp echhfS']/input")

        dest.send\_keys('new')

        res=self.driver.find\_elements(By.XPATH,"//\*/ul[@id='autoSuggest-list']/li")

        actions=ActionChains(self.driver)

        for each in res:

            if 'india' in each.text.lower():

                actions.move\_to\_element(each).click().perform()

                break

        sleep(1)

#########################

# Scrolling down of webpage #

#########################

Sometime we need to scroll down the webpage to perform some action on that elemenet.

If element is not visible and we do any operation on that then we will get exception.

We can scroll down in below way by using execute\_script() –

1. driver.execute\_script("window.scrollBy(**x-cord**,**y-cord**)","") --- scrolling to a co-ord
2. driver.execute\_script("window.scrollTo(0,document.body.scrollHeight)") -------- scroll till bottom

Example: - driver.execute\_script("window.scrollBy(x-cord,y-cord)","")

        for each in airlines:

            if 'canada'.lower() in each.text.lower():

                print('location of element is: ',each.location)

#scroll to coordinate (0,900)

                self.driver.execute\_script("window.scrollBy(0,900)","")

                #self.driver.execute\_script("window.scrollTo(0, document.body.scrollHeight);")

                sleep(2)

                each.click()

We have some other methods also-

1. scrollBy(x-coord, y-coord)
2. scroll(x-coord, y-coord)

* x-coord is the horizontal pixel value that you want to scroll by.
* y-coord is the vertical pixel value that you want to scroll by.

1. scrollBy(options)
2. scroll(options)

* **options** -A dictionary containing the following parameters: -- top, left

################################################################################# PyTest #

################################################################################

pytest is a framework that makes building simple and scalable tests easy.

pytest will run all files of the form *test\_\*.py or \*\_test.py* in the current directory and its subdirectories if file name is not specified

pytest [options] [file\_or\_dir] [file\_or\_dir] [...] ---------------- Usage

**Example:**

pytest test\_file1.py ---------- Will run all test cases from test\_file1.py.

pytest --------- Will run all test cases from all files of current and sub directories.

**Conventions for Python test discovery**

* If no arguments are specified, then collection starts from testpaths (if configured) or the current directory.
* In those directories, search for test\_\*.py or \*\_test.py files, imported by their test package name.
* From those files, collects test items (test cases):

1. *test prefixed test functions or methods outside of class*
2. *test prefixed test functions or methods inside Test prefixed test classes* (without an \_\_init\_\_ method)

**Advantages of pytest**

* Pytest can run multiple tests in parallel, which reduces the execution time of the test suite.
* Pytest has its own way to detect the test file and test functions automatically, if not mentioned explicitly.
* Pytest allows us to skip a subset of the tests during execution.
* Pytest allows us to run a subset of the entire test suite.
* Pytest is a free and open-source testing framework.

**Anatomy of test**

We can break any test case /suite in below four parts-

* Arrange ---- preparation part
* Act ---- Testing the functionality
* Assert ---- Comparing test result with expected result
* Cleanup ---- Closure part of test

**To learn the usage of the test fixtures [setUpClass & tearDownClass , setUpModule& tearDownModule]**

Circle.py file:

from math import  pi

class circle:

    def circle\_area(self,radius):

        if type(radius) not in [int, float]:

            raise TypeError("The radius must be a non-negative real number")

        if radius < 0:

            raise ValueError("The radius cannot be negative.")

        return pi \* (radius \*\* 2)

testverify.py file:

import unittest

from circle import circle

#make necessary imports

object1=None

#setUpModule-instantiates the object and returns the same.

def setUpModule():

    object1=circle()

    print("Module")

    return object1

#tearDownModule-uninstantiates the object.

def tearDownModule():

    object1=None

class Test(unittest.TestCase):

    #setupClass- creates a list of test data

    @classmethod

    def setUpClass(cls):

        cls.a=[2,3,10]

    def setUp(self):

        print("before test method")

    #test methods

    def test\_1(self):

        print("test1")

        print(setUpModule().circle\_area(self.a[0]))

    def test\_2(self):

        print("test2")

        print(setUpModule().circle\_area(self.a[1]))

    def test\_3(self):

        print("test3")

        print(setUpModule().circle\_area(self.a[2]))

    def tearDown(self):

        print("after test method ")

    @classmethod

    def tearDownClass(cls):

        print("in tear down class method")

#####################

# Pytest Reporting flags #

#####################

-r

show extra test summary info as specified by chars: **(f)**ailed, **(E**)rror, (**s)**kipped, **(x)**failed, **(X)**passed,**(p)**assed, **(P)**assed with output, **(a)**ll except passed **(p/P),** or **(A)**ll. (w)arnings are enabled by default (see --disable-warnings), 'N' can be used to reset the list. (default: 'fE').

-rA ---- Shows test summary for all test cases

-rf ----- Shows test summary for failed

--maxfail=num

exit after first num failures or errors.

--duration=N

how N slowest setup/test durations (N=0 for all).

--no-header disable header

-- no-summary disable summary

-q, -quite decrease verbosity.

########################

# Pytest general parameters #

########################

-k expression

only run tests which match the given substring expression. An expression is a python evaluable expression where all names are substring-matched against test names and their parent classes.

**Example 1**:

-k 'test\_method or test\_other' matches all test functions and classes whose name contains 'test\_method' or 'test\_other'

**Example 2:**

-*k 'not test\_method'* matches those that don't contain 'test\_method' in their names. -k 'not test\_method and not test\_other' will eliminate the matches.

-x

exit instantly on first error or failed test.

-marker

show markers (builtin, plugin and per-project ones).

-lf, -last-failed

rerun only the *tests that failed at the last run (or all if none failed)*

-v

Increase verbosity

####################################

# Generating pytest report in html format #

####################################

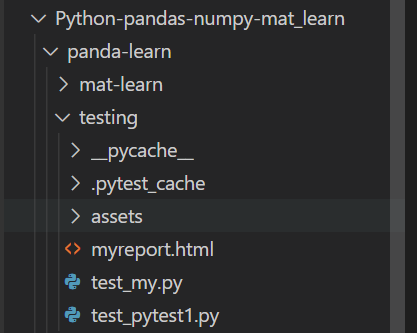
For generating report into html format then we need to use third party package 'pytest-html'.

For its usage check documentation.

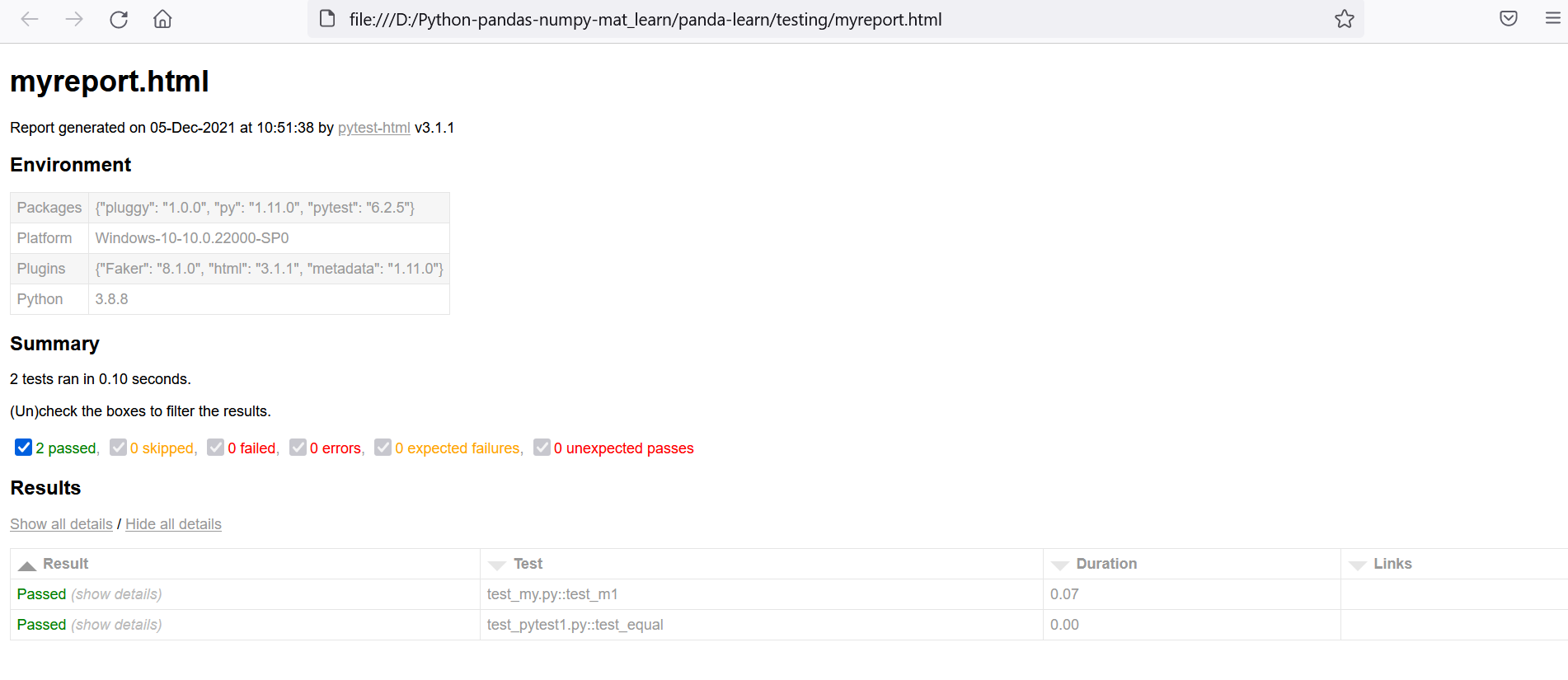
pytest [running options] –html='report\_file\_path.html' ----------- Generating report.

**Example**:

pytest --html="myreport.html"



Report



#################

# pytest exit codes #

#################

Running pytest can result in six different exit codes:

**Exit code 0:** All tests were collected and passed successfully

**Exit code 1:** Tests were collected and run but some of the tests failed

**Exit code 2:** Test execution was interrupted by the user

**Exit code 3:** Internal error happened while executing tests

**Exit code 4:** pytest command line usage error

**Exit code 5:** No tests were collected

###############################

# Running testcases in parallel mode #

###############################

For running test cases in parallel mode we need to install 'pytest-xdist' library.

Once it is installed then use -n flag when executing script.

pytest -n <number\_of\_test\_cases\_want\_to\_run\_in\_parallel> file\_name

Example:

pytest -n 10 test\_learn.py ---- It will run 10 testcases in parallel

###############

# PyTest Marker #

###############

Markers can be used to apply meta data to test functions (but not fixtures), which can then be accessed by fixtures or plugins. There are two types of markers in python.

Markers are used to set various features/attributes to test functions. Pytest provides many inbuilt markers such as xfail, skip and parametrize. Apart from that, users can create their own marker names

1. In built markers
2. Custom markers

###################

# How to use marker #

###################

We can use it with function (test function name) name.

@pytest.mark.marker\_name ----- for custom and in built both

def test\_testname():

#

**Note:**

If we are using any custom marker but not registered then will get warning but we can use it.

#####################

# Pytest inbuilt markers #

#####################

We have below inbuilt marker.

pytest.mark.filterwarnings(filter)

filter ----- String , A warning specification string, which is composed of contents of the tuple (action, message, category, module, lineno)

pytest.mark.skip(reason)

reason ------- String, Unconditional skipping and test function.

pytest.mark.skipif(comdtion,reason)

condition ---- conditional statement to decide skipping of test

reason ----- String, some message

This is for skipping a test function when some condition arrives.

pytest.mark.usefixtures(\*names)

name ------ name of fixture to use as string.

pytest.mark.xfail(condition=None, \*, reason=None, raises=None, run=True, strict=False)

This is used on those function which is suppose to raise any exception. If used on any function then that function must raise exception. Similar to @unitest.expectedFailure()

*condition (bool or str)* ---- Condition for marking the test function as xfail

*reason (str)* ------ Reason why the test function is marked as xfail.

*raises (Type[Exception])* ----- Exception subclass expected to be raised by the test function

*run (bool)* ----- If the test function should actually be executed.If False, the function will always xfail and will not be executed (useful if a function is segfaulting)

pytest.mark.parametarize(param1,param2..paramn,[(val1,val2..valn),…(val1,val2..valn)])

This is used for passing some values to test case. Test cases will execute for all sets of values.

param1,param2..paramn ----- These are parameters

[(val1,val2..valn),…(val1,val2..valn)] ---- These are list of tupple. It’s list of values for each param.

################

# Custom marker #

################

If we are using any custom marker then we need to register it first.

If we are using any marker which is not registered or not inbuilt then we will get warning.

Steps to register

1. Create pytest.ini file in current path or project
2. Add below entries into it

[pytest]

markers =

maker1 : “some description of marker 1”

maker2 : “some description of marker 2”

.

.

maker n : “some description of marker n”

**description of marker are optional.**

**Running tests with marker name**

Use -m flag to run any specific marker test function.

pytest -m “mymarker” ------ Run only that test fuction which have mymarker

**Example:**

pytest.init file

[pytest]

markers =

    smoke: 'this is smoke case'

    login:'this is login marker'

import py

import pytest

@pytest.mark.smoke #Custom smoke marker, it will run

def test\_m1():

    print('this is 1')

    assert 1==1

@pytest.mark.skip #Uncoditional skip, it will be skipped

def test\_login():

    print('logging in')

    assert 1==2

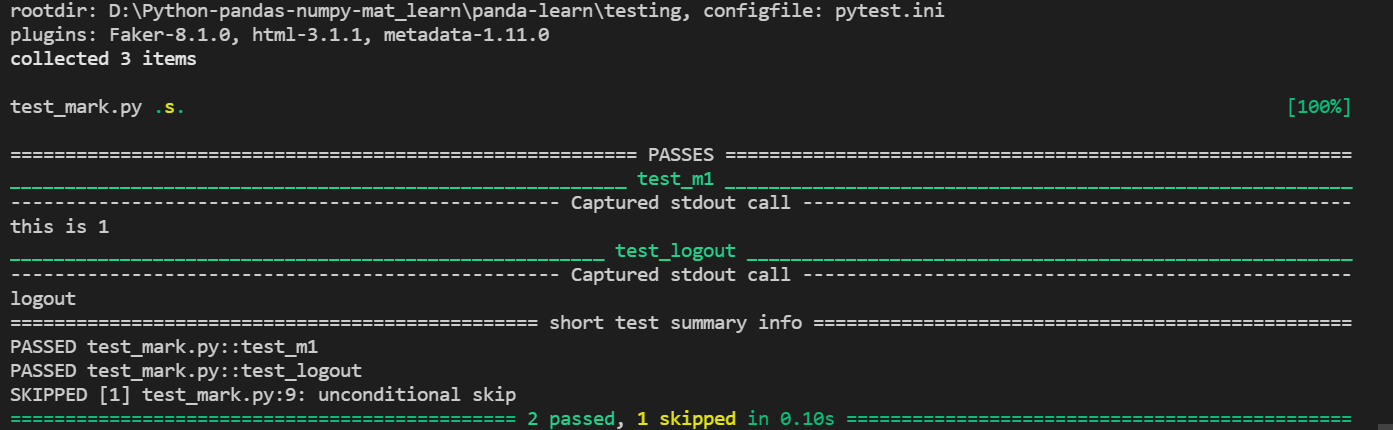
@pytest.mark.skipif(3<2,reason='1<2, skkiping it') #coditional skip, 3!<2 so will run

def test\_logout():

    print('logout')

    assert 2==2

Result:



############

# pytest.ini #

############

Pytest.ini is configuration file, using this file we can do lot of configurations. Check below link for all available configuration options-

<https://docs.pytest.org/en/6.2.x/reference.html#ini-options-ref>

Some common are-

**markers**

this is used for registering custom markers.

[pytest]

addopts = --strict-markers

*markers =*

*slow*

*serial*

**minversion**

Specifies a minimal pytest version required for running tests.

# content of pytest.ini

[pytest]

*minversion* = 3.0 # will fail if we run with pytest-2.8

**adopts**

Add the specified OPTS (*Space separated value*) to the set of command line arguments (pytest –help --- to check all command line argument) as if they had been specified by the user.

[pytest]

addopts = --maxfail=2 -rf # exit after 2 failures, report fail info

**filterwarnings**

Sets a list of filters and actions that should be taken for matched warnings. By default all warnings emitted during the test session will be displayed in a summary at the end of the test session.

[pytest]

filterwarnings =

error

ignore::DeprecationWarning

**Grouping of test cases:**

We can group any number of test cases in one group and can run those group of methods using custom marker. It help to running similar kind of tests one at a time.

**Step:**

* Decorate the test case function by any custom marker(marker\_name)
* Register that marker ----- Not mandatory (if not registered then will get warning but will work)
* Run them using -m flag --- pytest -m marker\_name file\_name

**Qeustion**: For below test cases, run method tes\_m1 and test\_m3 as one group

import pytest

def test\_m1():

    assert 1==1

def test\_m2():

    assert True==True

def test\_m3():

    assert 'admin'=='admin'

def test\_m4():

    assert 'name'=='rakesh'

Solution:

Mark those two methods by any marker name

import pytest

@pytest.mark.starter

def test\_m1():

    assert 1==1

def test\_m2():

    assert True==True

@pytest.mark.starter

def test\_m3():

    assert 'admin'=='admin'

def test\_m4():

    assert 'name'=='rakesh'

run it using -m flag.

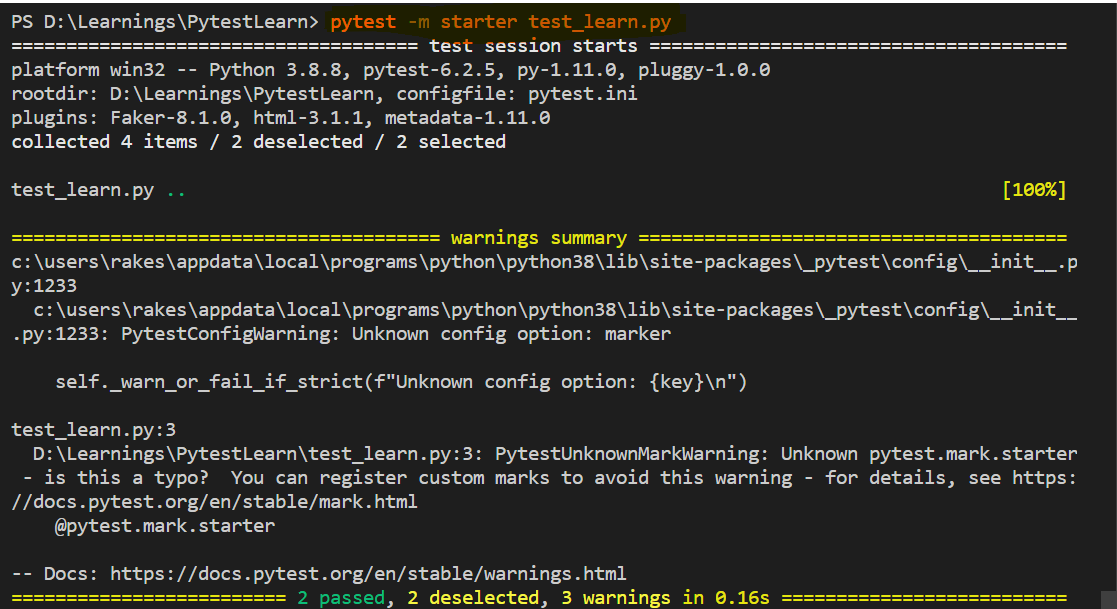
pytest -m starter test\_learn.py

**Note:**

We have registered the marker so we will get some warning.

From output you can see it’s running only those methods which have marker 'starter'

Output-



################################################################################

Fixtures in pytest

################################################################################

Fixtures are nothing but a function (any python function) decorated with @pytest.fixtures decorator.

Fixtures are *used to feed some data to the tests* such as database connections, URLs to test and some sort of input data.

Advantage of fixture is that it provides code reusability

Fixtures are of two types ---- 1.) Built-in fixtures 2.) Custom fixtures

#################

# Creating fixture #

#################

Creating a fixture is nothing but a python function decorated with @pytest.fixture decorator.

@pytest.fixture

def login():

#

**Note:**

1. Here name of function doesn’t need to be in test\_\* or \*\_test format.
2. We can return data from fixture also

#################

#How to use fixture#

#################

Use the fixture name as argument in test function. e.g-

**def test\_login(login):** #login is fixture

#

Important points:

1. A fixture can use/request another fixture
2. A test function can take/accept/request any number of fixture
3. Fixtures are reusable
4. Fixtures can be request more than once per test (return values are cached)
5. autouse fixture don’t need to requested by any function.

**Scope of fixture**

We can declare the scope of fixture using scope attribute while fixture creation.

@pytest.fixture(scope="specify\_the\_scope")

def login():

**Type of scope:**

We can have below 5 types of scope-

* function
* module
* class
* package
* session

**function:** when the scope is set to function, then the object will be teardown immediately after the requesting test function is terminated. When another test function calls that fixture it will rerun and create a new object. by default, the fixtures will have a function scope.

**module:** when the first time a function calls the fixture it will create the object and save it for the usage of test functions in the same module. So if any of the test functions in the same module calls that fixture the cached object will be returned. The object will be torn down after all the tests in that module completes.

**class:** the fixture will be executed once per class, it will be reused by all the test functions of that same test class. It will tear down after all the test functions from that class are completed.

**package:** fixture will be executed on first request and cached until all the test functions in that class get executed, and will be torn down after that.

**session:** this is the broad scope in pytest fixture, whenever we call pytest it is known as a session. So fixtures with session scope will be executed and cached on the first request and they will be reused until all the tests are completed.

@pytest.fixture(scope='class')

def get\_driver():

    return 1

class Test\_Class:

    def test\_m1(self,get\_driver):

        assert get\_driver==1

    def test\_m2(self,get\_driver):

        assert bool(get\_driver)==True

######################

# Creating custom fixture#

######################

Let’s take a look on all argument that a function can take while creating fixture.

@**fixture**(fixture\_function: \_FixtureFunction, \*, scope: Union[\_Scope, Callable[[str, Config], \_Scope]] = 'function', params: Optional[Iterable[object]] = None, autouse: bool = False, ids: Optional[Union[Iterable[Union[None, str, float, int, bool]], Callable[[Any], Optional[object]]]] = None, name: Optional[str] = None)

@fixture(fixture\_function: None = None, \*, scope: Union[\_Scope, Callable[[str, Config], \_Scope]] = 'function', params: Optional[Iterable[object]] = None, autouse: bool = False, ids: Optional[Union[Iterable[Union[None, str, float, int, bool]], Callable[[Any], Optional[object]]]] = None, name: Optional[str] = 'None')

Description of parameters are –

* scope

The scope for which this fixture is shared; one of "function" (default), "class", "module", "package" or "session".

* Params

An optional list of parameters which will cause multiple invocations of the fixture function and all of the tests using it. The current parameter is available in request.param.

* autouse

If True, the fixture func is activated for all tests that can see it. If False (the default), an explicit reference is needed to activate the fixture.

* name

The name of the fixture. This defaults to the name of the decorated function.

* ids

List of string ids each corresponding to the params so that they are part of the test id. If no ids are provided they will be generated automatically from the params.

Example:

import pytest

#Create a fixture

@pytest.fixture()

def inject\_dependency():

    # one can write here the code for connecting to Excel or a database

    print("inside pytest fixture")

#Create fixture with module scope will execute only once when this file runs

@pytest.fixture(scope='module')

def module\_inject\_dependency():

    print("inside module scoped fixture")

#Using fixture in test case

def test\_fixture\_demo(inject\_dependency,module\_inject\_dependency):

    print("executing fixture demo test case")

#Using fixture in test case

def test\_fixture\_demo\_1(inject\_dependency,module\_inject\_dependency):

    print("executing fixture demo test 2")

Note:

1. inject\_dependency will be executed twice as it is mapped to two test methods and the default "scope" is "function".
2. module\_inject\_dependency() will be executed once because the "scope" is set to "module".

**Parameterization of test fixture or testing function: -**

Parameterization is nothing but passing some input variable and associated values to test.

@pytest.mark.parametrize("var1,var2,..varn",[(val1,val2,..valn),[(val1,val2,..valn),.(val1,val2,..valn)])

def test\_add(arg1, arg2, resultoutput):

assert var1+var2+..varn== varn,"failed"

**Note:**

* var1,var2,…varn are variable that we are passing
* (val1, val2, .. valn) are set of value of variable var1, var2..varn.
* If we have to pass multiple values sets of each variable then pass them as list of set.
* A function can have any number of parameterizations.
* Test function will execute for each combination of input.

Example: -

test\_learn.py

import pytest

@pytest.mark.parametrize("num1,num2",[[1,2],[2,3],[5,6]])

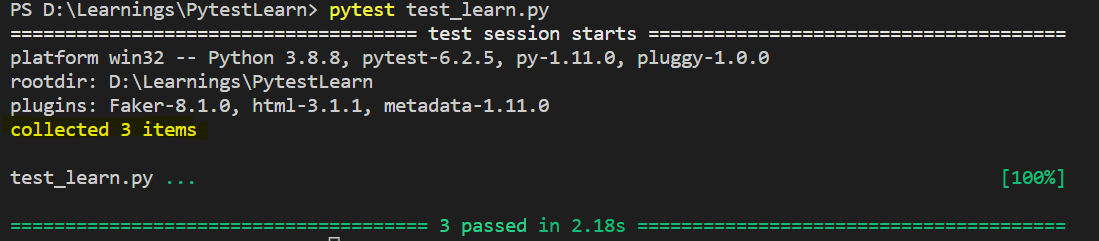
def test\_exce\_parametrization\_demo(num1,num2):

    print(num1,num2)

**Note:**

Above method will run three times b/c we have three sets of inputs.

Output:



**Setting up and tearing down in pytest**

For obtaining this behaviour we should take use of **yield** keyword in fixture.

For setting up (like setup() ) we should use fixture and for tearing down (like teardown()) we should use yield keyword in fixture then

**@pytest.fixture(scope=’define\_score’)**

def test\_setup\_fixture():

#setup part

**yield var\_name**

#teardown pat

Example:

import pytest

from selenium import webdriver

from time import sleep

@pytest.fixture(scope='class')

def get\_driver():

    driver=webdriver.Firefox(executable\_path="geckodriver.exe")

    yield driver

    #Now teardown

    driver.close()

    driver.quit()

class Test\_Class:

    def test\_m1(self,get\_driver):

        print('testing function 1')

        get\_driver.get("https://www.youtube.com/")

        assert 1==1

    def test\_m2(self,get\_driver):

        print('testing function 1')

        assert 2==2

**Note:**

First it will run the setup part then test methods from Test\_Class then finally it will close and quite the browser.

**Generating HTML Report**

For generating html report, we must install 'pytest-html' library in python and run with below flag-

--html='report\_file\_path.html'

**Example:**

pytest -v -s --html=report.html test\_learn.py

**###################**

**# Page Object Model #**

**###################**

Page Object Model is a design pattern where the core focus is on reducing code duplication, reusablity and minimization of the effort involved in code update/maintenance.

**Page Object Element (Page Class/Page Object)** – The Page Class is an object repository for the WebElements/Web UI Elements of the web-pages under test. It also contains an implementation of the interfaces/methods to perform operations on these web elements. It is loosely based on the fundamentals of Object-Oriented Programming.

**Test Cases** – As the name suggests, test cases contain the implementation of the actual test scenarios. It uses page methods/methods in the page class to interact with the page’s UI elements. If there is a change in the UI of the web page, only the Page Class needs to be updated, and the test code remains unchanged.

**Advantage of POM:**

* Increased Reusability
* Improved Maintainability
* Minimal impact due to UI changes
* Integration with multiple test frameworks

Generally, POM follow below structure-

Project-Directory

|--------- Src

|--------- PageObject

|--------- Pages

|--------- \*Page.py (Implementation of methods that make use

of the respective Locators declared in Locators.py)

(Contains all methods for specific page testing)

|--------- Locators.py (file contains all locators required for AUT in use)

|--------- TestBase

|--------- WebDriverSetup.py

|--------- Test

|--------- Scripts

|--------- test\_\*.py (Implementation of test code)(There should be 1:1

mapping of \*Page.py and test\_\*.py as it helps in making

the code more modular)

(contains all test\_cases for testing , calls methods of pages File)

|--------- TestSuite

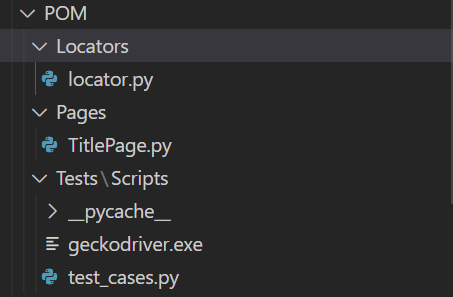
|--------- TestRunner.py (contains TestSuite, which is a collection of test

cases)

Example: -- Using POM concept find the title of youtube page.

Note:

Pages,Locator,Tests these are all packages not folder, do in Eclipse



**locator.py file**

class My\_Locator:

    #xpath for title

    title\_xpath='//\*/title[1]'

TitlePage.py

from POM.Locators.locator import My\_Locator

from selenium.webdriver.common.by  import By

class Title:

    def \_\_init\_\_(self,driver):

        self.driver=driver

    def get\_title(self):

        self.driver.maximize

        title=self.driver.find\_element(By.XPATH,My\_Locator.title\_xpath)

        return title.text

test\_cases.py

import pytest

from POM.Pages.TitlePage import Title

from selenium import webdriver

#Fixture for creating driver

@pytest.fixture(scope='class')

def driver\_initialize():

    print('starting driver fixture')

    driver=webdriver.Firefox(executable\_path="geckodriver.exe")

    yield driver

    print('now closing driver')

    driver.close()

    driver.quit()

#Using driver to get the title and doing assertion

def test\_get\_title(driver\_initialize):

    driver=driver\_initialize.get('https://www.youtube.com/')

#Create the instance of Title class

    ttl=Title(driver)

#call get\_title function to get the title of page

    title\_text=ttl.get\_title()

    assert title\_text=="YouTube"

**###############################################################################**

**# More on Unittest in python #**

**###############################################################################**

This class is used to compile information about which tests have succeeded and which have failed.

TestCase and TestSuite classes ensure that results are properly recorded; test authors do not need to worry about recording the outcome of tests.

TestResult instances have the following attributes that will be of interest when inspecting the results of running a set of tests:-

**errors**

A list containing 2-tuples of TestCase instances and strings holding formatted tracebacks. Each tuple represents a test which raised an unexpected exception.

**failures**

A list containing 2-tuples of TestCase instances and strings holding formatted tracebacks. Each tuple represents a test where a failure was explicitly signalled using the TestCase.assert\*() methods

**skipped**

A list containing 2-tuples of TestCase instances and strings holding the reason for skipping the test.

**expectedFailures**

A list containing 2-tuples of TestCase instances and strings holding formatted tracebacks. Each tuple represents an expected failure or error of the test case.

**unexpectedSuccesses**

A list containing TestCase instances that were marked as expected failures, but succeeded.

################################################################################

Pytest Function

################################################################################

In pytest we have some function which are generally used in testing purpose. We can check them all on below link.

<https://docs.pytest.org/en/6.2.x/reference.html#functions>

approx(expected, rel=None, abs=None, nan\_ok: bool = False)

Asserts that two numbers (or two sets of numbers) are equal within some tolerance.

It can be used with many data types --- dictionary, list, numpy array etc

Due to the intricacies of floating-point arithmetic, numbers that we would intuitively expect to be equal are not always, e.g ---- .1+.2==.3 #false in python

**with scalar/values-**

0.1 + 0.2 == approx(0.3)

**Sequence of values/data**

(0.1 + 0.2, 0.2 + 0.4) == approx((0.3, 0.6))

**Dictionary values**

{'a': 0.1 + 0.2, 'b': 0.2 + 0.4} == approx({'a': 0.3, 'b': 0.6})

**numpy array**

np.array([0.1, 0.2]) + np.array([0.2, 0.4]) == approx(np.array([0.3, 0.6]))

**numpy array against scalar**

np.array([0.1, 0.2]) + np.array([0.2, 0.1]) == approx(0.3)

fail(msg: str = '', pytrace: bool = True)

Explicitly fail an executing test with the given message.

**msg (str)** – The message to show the user as reason for the failure.

**pytrace (bool)** – If False, msg represents the full failure information and no python traceback will be reported.

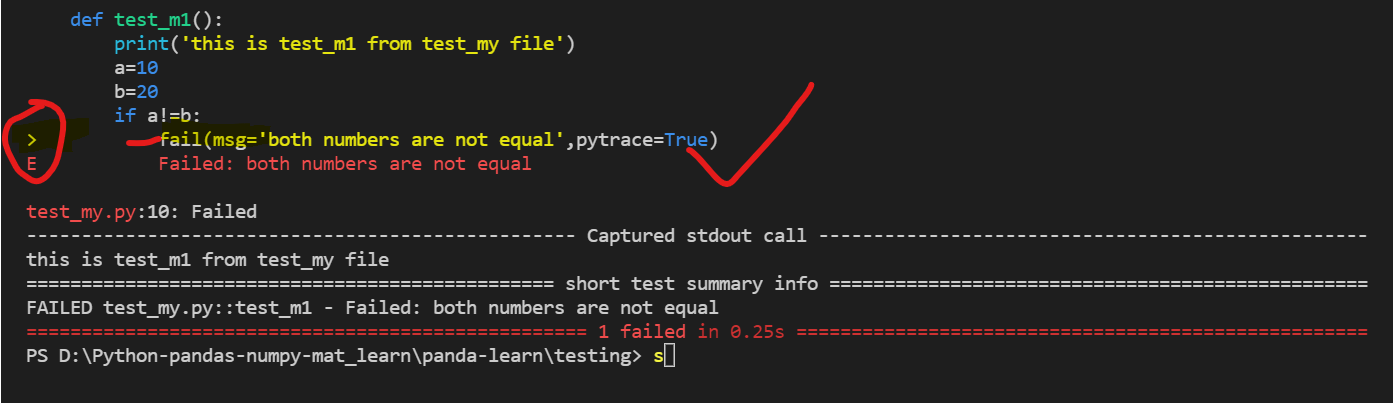
def test\_m1():

    a=10

    b=20

    if a!=b:

        fail(msg='both numbers are not equal',pytrace=True)



skip(msg[, allow\_module\_level=False])

Skip an executing test with the given message.

This function should be called only **during testing (setup, call or teardown)** or during collection by using the allow\_module\_level flag.

This function can be called in doctests as well.

**allow\_module\_level:** (bool) – Allows this function to be called at module level, skipping the rest of the module. Defaults to False.

**msg :** Message to give, str

importorskip(modname: str, minversion: Optional[str] = None, reason: Optional[str] = None)

Import and return the requested module modname, or skip the current test if the module cannot be imported.

**modname**: (str) – The name of the module to import.

**minversion**: (str) – If given, the imported module’s \_\_version\_\_ attribute must be at least this minimal version, otherwise the test is still skipped

**reason**: (str) – If given, this reason is shown as the message when the module cannot be imported.

xfail(reason: str = '')

Imperatively xfail an executing test or setup function with the given reason.

exit(msg: str, returncode: Optional[int] = None)

Exit testing process.

#########################################

# Creating test class #

#########################################

We can create any test class same as python class additional below naming convention:-

1. Name of class with start with ‘TEST’
2. All methods (for testing purpose) will start or end with ‘test’

e.g-

class TestClass:

        def test\_tc1(self): #self we need to pass as this will be passed by pytest

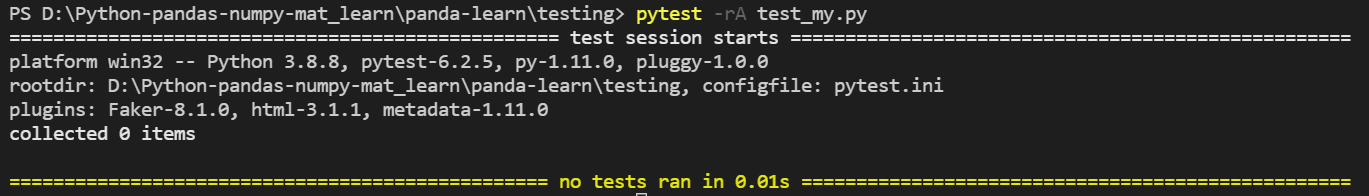
        assert 1==1 #self will be passed by default by pytest

    def test\_tc2(self):

        assert 2==1

**Note:**

1. For each method we need to use self as argument b/c pytest will pass one argument for those test method (test methods are instance method)
2. All of our test method should be instance methods else pytest will not capture them and will not be executed
3. class TestClass:
5. @classmethod
6. def test\_m3(cls): #defining test function to be class method
7. print('method m4')
8. assert 4==4
9. TestClass.test\_m3()



Test method (declared as class method) is not executed as we can see.

##############################

# Test setup and teardown #

##############################

We can do setup and teardown at class level, module level, method level.

**Module level setup/teardown**

If you have multiple test functions and test classes in a single module you can optionally implement the following fixture methods *which will usually be called once for all the functions:*

(gets call only once when file executes)

def setup\_module(module):

""" setup any state specific to the execution of the given module."""

def teardown\_module(module):

"""teardown any state that was previously setup with a setup\_module

method. At end of file execution

"""

**Class level setup/teardown**

Following methods are called at class level before and after all test methods of the class are called:

*Before each test method execution and after execution of class.*

@classmethod

def setup\_class(cls):

"""setup any state specific to the execution of the given class (which

usually contains tests).

"""

@classmethod

def teardown\_class(cls):

"""teardown any state that was previously setup with a call to

setup\_class.

"""

class TestClass:

    @classmethod

    def setup\_class(cls): #Should be called before file execution start

        print('steup\_class')

    @classmethod #should be called end of file execution

    def teardown\_class(cls):

        print('teardown\_class')

    def setup\_method(self, method): #should be called before of each method

        print('setup\_method')

    def teardown\_method(self, method): #should be called after of each method

        print('teardown\_method')

    def test\_tc1(self):

        print('tc1')

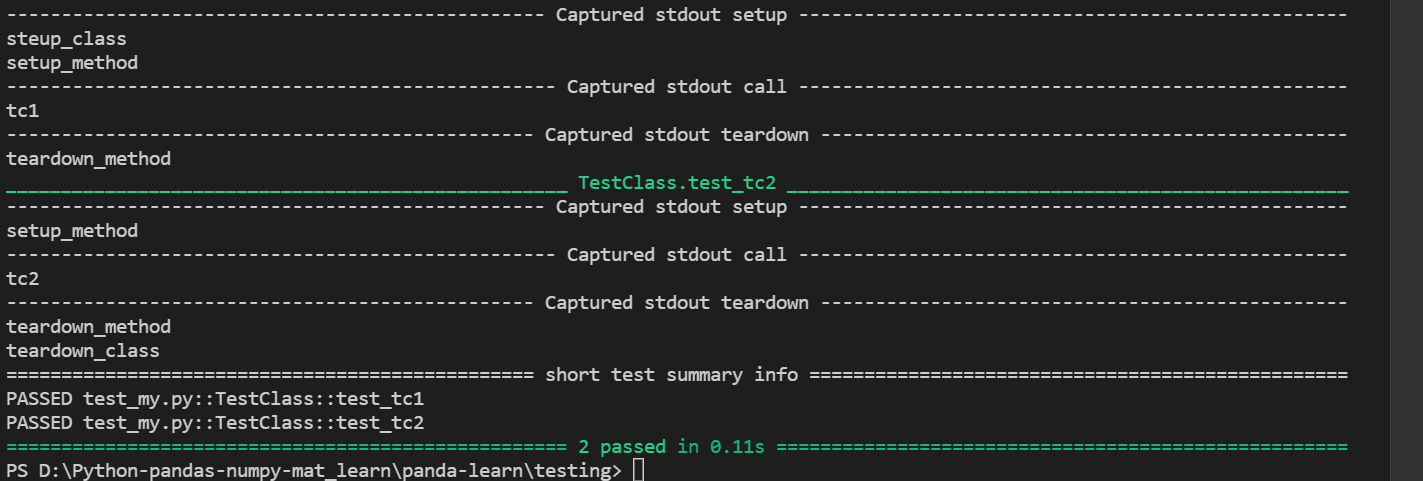
        assert 1==1

    def test\_tc2(self):

        print('tc2')

        assert 2==2

Output:



**Method and function level setup/teardown**

def setup\_method(self, method):

"""setup any state tied to the execution of the given method in a

class. setup\_method is invoked for every test method of a class.

"""

def teardown\_method(self, method):

"""teardown any state that was previously setup with a setup\_method

call.

################################################################################

# REST API Testing #

################################################################################

**JSON**

JSON (JavaScript Object Notation) is a popular data format used for representing structured data. It's commonly used in transmitting and receive data between a server and web application in JSON format

**JSON data can be in below format**

* A key-value pair. E.g. object, dictionary, hash tables, etc.
* A collection of values. E.g. List, array, etc

**NOTE:**

* **Key:** A key is always a string enclosed in quotation marks.
* **Value:** A value can be a string, number, boolean expression, array, or object.
* JSON data looks like python dictionary data types, JSON data are always kept in double quote but dictionary can be in single or double quote.
* We can check if data is a valid JSON data using loads().

**Json data types:**

In JSON, data type is considered on basis of values not keys and values must be one of the following data types

1. String
2. Number
3. Object (JSON object) --- kind of dict of dict, i.e values itself will be a JSON data
4. Array ---- values will be list/array type
5. Boolean
6. null

JSON object Example-

{

"employee":{ "name":"Neel", "age":28, "city":"Landon" }

}

Array Example-

{

"employees":[ "Neel", "Jhon", "Harry" ]

}

**Json String**

Strings in JSON must be written in double quotes.

{ "name":"John" }

**Number**

Numbers in JSON must be an integer or a floating point.

{ "age":30 }

**JSON Object**

A JSON object contains data in the form of key/value pair. The keys are strings and the values are the JSON types. Keys and values are separated by colon. Each entry (key/value pair) is separated by comma.

*The { (curly brace) represents the JSON object*

*{*

*"employee": {*

*"name": "sonoo",*

*"salary": 56000,*

*"married": true*

*}*

*}*

**Json Array**

The data enclosed into '[',']' (square bracket) represents the JSON array.

*A JSON array can have values and objects.*

e.g—

1. Simple JSON array

["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"]

2. Json array of json objects

[

{"name":"Ram", "email":"Ram@gmail.com"},

{"name":"Bob", "email":"bob32@gmail.com"}

]

**Boolean**

Booleans in JSON are either true or false

**Python data to JSON data types conversion tables.**



**Json data to python data conversion table**



**Parse JSON into Python**

**Deserialization**

This is process of converting JSON data into Python format. For this JSON have load() and loads() in python json library.

**Parse JSON into Python**

The json module makes it easy to parse JSON strings and files containing JSON object

We have two methods to part it

1. json.loads(json\_data) ------ from JSON variable to python dictionary
2. json.load(file\_object) -------- from JSON file to python

**json.loads()**

using this method, we can convert JSON string into python data type or python data in a variable

We can parse a JSON string using json.loads() method.

*The method returns a dictionary if input data is validate json data else will get ValueError exception*.

Using load() method we can check if input data is a valid input data.

import json

person = '{"name": "Bob", "languages": ["English", "Fench"]}'

person\_dict = json.loads(person)

# Output: {'name': 'Bob', 'languages': ['English', 'Fench']}

print( person\_dict)

NOTE:

Here person is json string that is converted into python dictionary

**json.load(file\_object)**

We can use this method to convert JSON data stored in file to python variable

We can use json.load() method to read a file containing JSON object.

import json

with open('path\_to\_file’) as f:

  data = json.load(f)

print(data)

**Serialization**

Process of converting python data type to JSON type is called serialization. For this in json we have dumps() and dump() methods.

**Converting Python to JSON string**

**Python(dict,list,tuple etc) Convert to JSON string**

We can convert any of python data types into equivalent data types, for this we have below two methods

1. json.dumps(python\_var) ---------- from python variable to JSON string
2. json.dump(file\_path) ------------

**json.dumps(python\_var)**

using dumps(pyth\_variable) we can convert any python variable into it’s equivalent json data types

mport json

person\_dict = {'name': 'Bob',

'age': 12,

'children': None

}

person\_json = json.dumps(person\_dict)

# Output: {"name": "Bob", "age": 12, "children": null}

print(person\_json)

More about the dumps() can be check on next page

**json.dump(python\_data,file\_object)**

this method is used to write python data into JSON file or converting from python JSON variable to JSON file

import json

person\_dict = {"name": "Bob",

"languages": ["English", "Fench"],

"married": True,

"age": 32

}

with open('person.txt', 'w') as json\_file:

  json.dump(person\_dict, json\_file)

Above program will convert python data of variable person\_dict into person.txt file in which data will be in JSON format.

More about json.dump() can be checked below

**json.dump(obj, skipkeys=False, allow\_nan=True, cls=None, indent=None, separators=None, default=None, sort\_keys=False, \*\*kw)**

1. If skipkeys is true (default: False), then dict keys that are not of a basic type (str, int, float, bool, None) will be skipped instead of raising a TypeError.
2. If indent is a non-negative integer or string, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0, negative, or "" will only insert newlines.
3. Separators are used to specify how key and value will be separated for custom separation

separators=("separator\_for\_key", "separation\_for\_value ")

1. sort\_keys parameter to specify if the result should be sorted or not, if TRUE then the output of dictionaries will be sorted by key

**json.dumps(obj, skipkeys=False, allow\_nan=True, cls=None, indent=None, separators=None, default=None, sort\_keys=False, \*\*kw)**

Serialize obj to a JSON formatted str using this conversion table. *The arguments have the same meaning as in dump().*

**Converting Python class objects into JSON**

We can use \_\_dict\_\_ attribute of python to convert into dictionary and then use dumps() to convert into JSON.

Obj--- python object for any class

Obj.\_\_dict\_\_ ------- this will convert into python dictiaory

dumps(Obj.\_\_dict\_\_) --------- JSON format

**Encoders and Decoders**

**Encoders**

**class json.JSONEncoder(\*, skipkeys=False, ensure\_ascii=True, check\_circular=True, allow\_nan=True, sort\_keys=False, indent=None, separators=None, default=None)**

If we do conversion from Python to JSON data types they are converted to their equivalent data types as listed below



**Decoders**

Supports the following objects and types by default

**class json.JSONDecoder(\*, object\_hook=None, parse\_float=None, parse\_int=None, parse\_constant=None, strict=True, object\_pairs\_hook=None)**



**http Status code in Django**

1XX ----- this is for informational, it’s range is from 100 to 199

2XX ----- this is for success, it range is from 200 to 299

3XX ----- this is re-directional, it range from 300 to 399

4XX ----- Client Error, ranges from 400 to 499 (e.g --- page not found)

5XX ----- Server error

################################################################################

# **requests #**

################################################################################

Requests is a Python library that we can use to send all kinds of HTTP requests. It is an easy-to-use library with a lot of features ranging from passing parameters in URLs to sending custom headers and SSL Verification

**CRUD methods of requests**

Create --- post

Read ---- get

Update --- put/patch

Delete --- delete

requests has below methods-

|  |  |
| --- | --- |
| [delete(url, args)](https://www.w3schools.com/python/ref_requests_delete.asp) | Sends a DELETE request to the specified url |
| [get(url, params, args)](https://www.w3schools.com/python/ref_requests_get.asp) | Sends a GET request to the specified url  It returns response object |
| [head(url, args)](https://www.w3schools.com/python/ref_requests_head.asp) | Sends a HEAD request to the specified url |
| patch(url, data, args) | Sends a PATCH request to the specified url |
| [post(url, data, json, args)](https://www.w3schools.com/python/ref_requests_post.asp)  *post(*url*, data={*key*:* value*}, json={*key*:* value*},* args*) --- syntax*  arguments means zero or more named arguments | Sends a POST request to the specified url  It returns response object. |
| put(url, data, args) | Sends a PUT request to the specified url |
| request(method, url, args) | Sends a request of the specified method to the specified url |

**Belw are some parameters accepted by each methods**

* url --- URL where request will be sent
* data----Optional. A dictionary, list of tuples, bytes or a file object to send to the specified url. (anything send as data will be fetched from body .this corresponds to body part of response)
* headers --- Optional. A dictionary of HTTP headers to send to the specified url. Default None (see how to use in token authentication)
* timeout ---A number, or a tuple, indicating how many seconds to wait for the client to make a connection and/or send a response. default None
* cookies ---- Optional. A dictionary of cookies to send to the specified url.Default None
* cert ---- Optional. A String or Tuple specifying a cert file or key.Default None

**get() method**

requests.get(url, params={key: value}, args)

args refers to zero or more of the named arguments in the parameter table below.

Parameter (params) values:

There are many parameters few of them are-

|  |  |
| --- | --- |
| *url* | Required. The url of the request |
| Params | Optional. A dictionary, list of tuples or bytes to send as a query string. Default None it goes with in form of – url?query\_string  Final url will – url?dict=value – If params is a dictionary |
| allow\_redirects | Optional. A Boolean to enable/disable redirection. |
|  | Default True (allowing redirects) |
| Auth | Optional. A tuple to enable a certain HTTP authentication. |
|  | Default None |
| Cert | Optional. A String or Tuple specifying a cert file or key. |
|  | Default None |
| Cookies | Optional. A dictionary of cookies to send to the specified url. |
|  | Default None |
| Headers | Optional. A dictionary of HTTP headers to send to the specified url. |
|  | Default None |
| Proxies | Optional. A dictionary of the protocol to the proxy url. |
|  | Default None |
| Stream | Optional. A Boolean indication if the response should be immediately downloaded (False) or streamed (True). |
|  | Default False |
| Timeout | Optional. A number, or a tuple, indicating how many seconds to wait for the client to make a connection and/or send a response. |
|  | Default None which means the request will continue until the connection is closed |
| Verify | Optional. A Boolean or a String indication to verify the servers TLS certificate or not. |
|  | Default True |

Questions:

For link - <https://reqres.in/api/users> pass the query string 'page=2' and make a get request.

Solution:

def test\_get\_data():

    URL="https://reqres.in/api/users"

    p={'page':2}

    res=requests.get(URL,params=p) #output will be - https://reqres.in/api/users?page=2

    print(res.url)

    assert res.status\_code==200

    print(res.status\_code)

**Note:**

We are passing dictionary in for params then finally url will be be – url?key=value&key1=value1..

This behaviour we can see in above code.

**post() method**

requests.post(url, data={key: value}, json={key: value}, args)

args means zero or more of the named arguments.

**Parameter values are:-**

|  |  |
| --- | --- |
| *url* | Required. The url of the request |
| data | Optional. A dictionary, list of tuples, bytes or a file object to send to the specified url |
| json | Optional. A JSON object to send to the specified url |
| files | Optional. A dictionary of files to send to the specified url |
| allow\_redirects | Optional. A Boolean to enable/disable redirection. |
|  | Default True (allowing redirects) |
| Auth | Optional. A tuple to enable a certain HTTP authentication. |
|  | Default None |
| Cert | Optional. A String or Tuple specifying a cert file or key. |
|  | Default None |
| Cookies | Optional. A dictionary of cookies to send to the specified url. |
|  | Default None |
| Headers | Optional. A dictionary of HTTP headers to send to the specified url. |
|  | Default None |
| Proxies | Optional. A dictionary of the protocol to the proxy url. |
|  | Default None |
| Stream | Optional. A Boolean indication if the response should be immediately downloaded (False) or streamed (True). |
|  | Default False |

**Methods and attributes of http response object**

Response is obtained whenever we send request using get()/post() to server. It has below methods/attributes

Some common methods or attributes of response objects are:-

response.json() ----- deserialization of json object(if result is in JSON format else error)

response .status\_code ---- returns status that indicates status

response .url --- returns url of response

response .close() ---- close connection to server

response.text ----- to get the content of response, in Unicode format

response.content --- to get the content of response in byte format

| **Method** | **Description** |
| --- | --- |
| [response.headers](https://www.geeksforgeeks.org/response-headers-python-requests/) | response.headers returns a dictionary of response headers. |
| [response.encoding](https://www.geeksforgeeks.org/response-encoding-python-requests/) | response.encoding returns the encoding used to decode response.content. |
| [response.elapsed](https://geeksforgeeks.org/response-elapsed-python-requests/) | response.elapsed returns a timedelta object with the time elapsed from sending the request to the arrival of the response. |
| [response.close()](https://www.geeksforgeeks.org/response-close-python-requests/) | response.close() closes the connection to the server. |
| [response.content](https://www.geeksforgeeks.org/response-content-python-requests/) | response.content returns the content of the response, in bytes. |
| [response.cookies](https://www.geeksforgeeks.org/response-cookies-python-requests/) | response.cookies returns a CookieJar object with the cookies sent back from the server. |
| [response.history](https://www.geeksforgeeks.org/response-history-python-requests/) | response.history returns a list of response objects holding the history of request (url). |
| [response.is\_permanent\_redirect](https://www.geeksforgeeks.org/response-is_permanent_redirect-python-requests/) | response.is\_permanent\_redirect returns True if the response is the permanent redirected url, otherwise False. |
| [response.is\_redirect](https://www.geeksforgeeks.org/response-is_redirect-python-requests/) | response.is\_redirect returns True if the response was redirected, otherwise False. |
| [response.iter\_content()](https://www.geeksforgeeks.org/response-iter_content-python-requests/) | response.iter\_content() iterates over the response.content. |
| [response.json()](https://www.geeksforgeeks.org/response-json-python-requests/) | response.json() returns a JSON object of the result (if the result was written in JSON format, if not it raises an error). |
| [response.url](https://www.geeksforgeeks.org/response-url-python-requests/) | response.url returns the URL of the response. |
| [response.text](https://www.geeksforgeeks.org/response-text-python-requests/) | response.text returns the content of the response, in unicode. |
| [response.status\_code](https://www.geeksforgeeks.org/response-status_code-python-requests/) | response.status\_code returns a number that indicates the status (200 is OK, 404 is Not Found). |
| [response.request](https://www.geeksforgeeks.org/response-request-python-requests/) | response.request returns the request object that requested this response. |
| [response.reason](https://www.geeksforgeeks.org/response-reason-python-requests/) | response.reason returns a text corresponding to the status code. |
| [response.raise\_for\_status()](https://www.geeksforgeeks.org/response-raise_for_status-python-requests/) | response.raise\_for\_status() returns an HTTPError object if an error has occurred during the process. |
| [response.ok](https://www.geeksforgeeks.org/response-ok-python-requests/) | response.ok returns True if status\_code is less than 200, otherwise False. |
| [response.links](https://www.geeksforgeeks.org/response-links-python-requests/) | response.links returns the header links. |

**HTTP GET method**

* HTTP GET is used to retrieve a resource which may be an HTML document, an image, an XML file, etc.
* If the resource is found, it returns HTTP response code 200 along with the XML or JSON content. If not found, HTTP response code 404 is returned.
* In REST services, using GET operation, we can fetch multiple properties like status code, content, header, etc. from a response. In this section, we will see how to work with HTTP GET methods.
* The response object will contain all the data sent by the server. Different properties of the response object are used to get different parts of the response.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1XX** ----- this is for informational, it’s range is from 100 to 199  **2XX** ----- this is for success, it range is from 200 to 299  **3XX** ----- this is re-directional, it range from 300 to 399  **4XX** ----- Client Error, ranges from 400 to 499 (e.g --- page not found)  **5XX** ----- Server error | |  |  | | --- | --- | | **Status Codes and their meanings** | | | Status code | Status Message | | 200 | OK | | 201 | Created | | 400 | Bad request | | 401 | Unauthorized | | 404 | Resource not found | | 405 | Method not allowed | | 501 | Internal server error | |

################################

# GET vs POST # ------- asked in interview

################################

|  |  |
| --- | --- |
| GET | POST |
| > Get request is not secured because data is exposed in URL bar  > In GET request **data is sent in header**  > In case of Get request, only **limited amount of data can be sent** because data is sent in header.  >GET request is generally for fetching data from server. | > Post request is secured because data is not exposed in URL bar.  >In **POST data is sent in body**  > In case of post request**, large amount of data can be sent** because data is sent in body.  >POST request is generally for send data to server. |

**JSON parsing for GET method**

* If you take a look at the response, it is actually serialized JSON content. Now we can deserialize this using JSON.loads() method as learned earlier in this course.
* But there is a simpler way to do this, i.e, using .json() method.
* .json() method returns a dictionary type object. So, you can access the values using the key.

Example:

import requests,json

import requests,json

def test\_EDU\_Bank():

    URL="some\_url"

    response=requests.get(url=URL)

    response\_data=response.json()

    print(response\_data) #here data will be converted into python form

    status\_code\_json=response.status\_code

    assert status\_code\_json==200

    headers\_json=response.headers

    print(headers\_json)

    content\_type=headers\_json['Content-type'].split(";")[0]

    assert content\_type=="application/json"

**Question:**

Make a post request to url <https://reqres.in/api/users> with below dictionary data.

{"name": "morpheus","job": "leader"}

**Solution**

def test\_get\_data():

    URL="https://reqres.in/api/users"

    dict\_data={"name": "morpheus","job": "leader"}

    my\_data=dumps({"name": "morpheus","job": "leader"})

    resp=post(URL,data=my\_data)

    print(resp.status\_code)

    print(resp.json())

**Question:**

Read the json data from a file (say – data.sjon) and make a post request to above url with json data read from file but we have to send data in dictionary format.

Method 1: ---

def test\_get\_data():

    URL=<https://reqres.in/api/users>

json\_data=open(‘data.json’,’r’).read()

dict\_data=loads(json\_data)

    resp=post(URL,data=dict\_data)

    print(resp.status\_code)

    print(resp.json())

Method 2: -

def test\_get\_data():

    URL=<https://reqres.in/api/users>

dict\_data=load(data.json)

    resp=post(URL,data=dict\_data)

    print(resp.status\_code)

    print(resp.json())

################################################################################

# Selenium Interview Question #

################################################################################

**What are the testing types supported by Selenium, explain them-**

Functional Testing

Regression Testing

Sanity Testing

Smoke Testing

Responsive Testing

Cross Browser Testing

UI testing (black box)

Integration Testing

**Functional Testing**

This type of testing is done to determine if a feature or system functions properly without issues.

It checks the system at different levels to ensure that all scenarios are covered and that the system does what it’s supposed to do. It mainly focuses on checking functionality is working or not.

**Example:**

Create account, delete account, modify user details etc in any website. It doesn’t matter how slow or fast website is processing.

**Performance Testing:**

As its name indicates, performance tests are done to measure how well an application is performing. It is done by tools like – Jmeter. There are two main sub-types for performance testing-

* Load Testing
* Stress Testing

**Load Testing**

Load testing is done to verify how well the application works under different defined loads (usually a particular number of users connected at once)

**Stress Testing**

Stress testing is done to verify how well the application works under stress (or above the maximum supported load).

**Regression testing**

This testing is generally done after a change, fix or feature addition.

To ensure that the change has not broken any of the existing functionality, some already executed tests are executed again.

The set of re-executed tests can be full or partial and can include several different types, depending on the application and development team

**Implicit and Explicit wait in Selenium**

**Implicit wait:**

An implicit wait is a condition-less wait command in Selenium that applies to all element on webpage. Implicit Wait stays in place for the entire duration for which the browser is open.

Using implicit wait we can tell Selenium that we would like it to wait for a certain amount of time before throwing an exception that it cannot find the element on the page or the page is not loaded or the javascript execution is not finished.

**Explicit wait:**

An explicit wait is a conditional wait strategy in Selenium in other words you wait until the condition you specified becomes true or the time duration has elapsed.

**Explain the difference between single (/) and a double slash (//) in X-path?**

Single slash (/) allows the creation of ‘absolute’ path expressions or locating element using absolute path, while double slash (//) allows the creation of ‘relative’ path expressions

**Is selenium Web Driver Automation tool?**

No it's not aumotation tool, it's a framework or set of class which is used to automate a web application testing

**What are all locators in selenium?**

**How Synchronization is achieved in selenium**

It is achieved by using implicit and explicit wait.

**How can we check if control is enabled.**

We can check it using is\_enabled() in below way-

web\_element.is\_enabled()

**How to get the value of a attribute of web element.**

element.get\_attribute('attribute') ---- finding value of a attribute of web element.

**What are all possible ways to find text of web element?**

We can find web element text context using below two ways-

element.text

element.get\_attribute('InnerText')

**What are all possible ways of selecting values from dropdown**

We can select element from dropdown using below ways-

select\_by\_index()

select\_by\_value()

select\_by\_visible\_text()

**What are all navigational commands in selenium**

We have below navigation command in selenium

driver.forward() --- to move forward on webpage

driver.back() --- To move backward on current page

driver.get() --- To open any url

driver.switch\_to.frame(frame\_handler)

driver.switch\_to.alert

driver.switch\_to.window(window\_handler)

**What is difference between find\_element and find\_elements**

find\_element --- Returns the first matching element. If not found then NoSuchElement Exception.

find\_element --- Return all matching element in list, no exception if element is not found.

**What is difference between driver.close() and driver.quite()**

**driver.close() ---** it closes the current browser window in focus.

**driver.quite()** ---- It closes all of the browser windows and ends the WebDriver session

**How to get screenshot using selenium**

driver.get\_screenshot(path) ---- saves as PNG format in specified path.

**Switch class in selenium**

Switch class are used to switch between the browsers windows, alert, frames etc in selenium.

We can use below method for switching –

* switch\_to.frame(frame\_hanlder)
* switch\_to.alert
* switch\_to.window(window\_handler)

**What are all ways to enter/send text to a element?**

We can send text/value to a web element in below two ways-

1. web\_element.send\_key()
2. action.send\_send\_key\_to\_element(element,value) --- action is ActionChain class instance

**How to get the current url of page?**

driver.current\_ulr()