

**1. `import pandas as pd`:**

- Imports the Pandas library and aliases it as `pd`. Pandas is used for data manipulation and analysis, especially for handling data in tabular form (DataFrames).

**2. `import time`:**

- Imports the `time` module, which provides various time-related functions. In this script, it is used for introducing delays (e.g., `time.sleep(3)`) to wait for content to load.

**3. `from bs4 import BeautifulSoup`:**

- Imports the `BeautifulSoup` class from the `bs4` (Beautiful Soup) library. Beautiful Soup is used for web scraping and parsing HTML or XML documents.

**4. `from selenium.webdriver.common.by import By`:**

- Imports the `By` class from the `selenium.webdriver.common.by` module. This class provides mechanisms for locating elements on a web page.

**5. `from selenium.webdriver.common.keys import Keys`:**

- Imports the `Keys` class from the `selenium.webdriver.common.keys` module. It provides special keys (like ENTER, SHIFT, etc.) that can be used in Selenium interactions.

**6. `from selenium.webdriver.support.ui import WebDriverWait`:**

- Imports the `WebDriverWait` class from the `selenium.webdriver.support.ui` module. This class is used for waiting until certain conditions are met before proceeding with the execution.

**7. `from selenium.webdriver.support import expected\_conditions as EC`:**

- Imports the `expected\_conditions` module from `selenium.webdriver.support` and aliases it as `EC`. It contains a set of predefined conditions that can be used with `WebDriverWait` for waiting.

**8. `from selenium.webdriver.chrome.service import Service as ChromeService`:**

- Imports the `Service` class from the `selenium.webdriver.chrome.service` module and aliases it as `ChromeService`. It represents the ChromeDriver service.

**9. `from selenium.webdriver.chrome.options import Options`:**

- Imports the `Options` class from the `selenium.webdriver.chrome.options` module. It is used for configuring Chrome options, such as running the browser in headless mode.

**10. `from selenium import webdriver`:**

- Imports the `webdriver` module from the `selenium` package. This module provides a way to instantiate and control web browsers, in this case, the Chrome browser.

**11. `import re`:**

- Imports the `re` module, which provides support for regular expressions. Regular expressions are used for pattern matching and extraction of information from text.

**12. `from selenium.common.exceptions import NoSuchElementException, TimeoutException`:**

- Imports specific exceptions (`NoSuchElementException` and `TimeoutException`) from the `selenium.common.exceptions` module. These exceptions are used to handle cases where an expected element is not found or a timeout occurs during the waiting process.



**1. `def initialize\_driver(headless=True):**

- This line defines a function named `initialize\_driver` that takes an optional argument `headless`, which is set to `True` by default.

**2. `chrome\_options = Options()`:**

- Creates an instance of the `Options` class from the `selenium.webdriver.chrome.options` module. This class is used to configure options for the Chrome browser.

**3. `if headless:`:**

- Checks if the `headless` argument is `True`. If `True`, it indicates that the browser should run in headless mode.

**4. `chrome\_options.add\_argument('--headless')`:**

- Adds the argument `--headless` to the Chrome options. This argument is used to run the browser in headless mode, meaning it runs without a graphical user interface.

**5. `headers = {...}`:**

- Defines a dictionary named `headers` containing a User-Agent string. The User-Agent string simulates the information sent by a web browser when making requests.

**6. `chrome\_options.add\_argument(f'user-agent={headers["User-Agent"]}')`:**

- Adds a custom User-Agent string to the Chrome options. This can be useful for avoiding bot detection or mimicking a specific browser during web scraping.

**7. `chrome\_service = ChromeService(executable\_path='C:/Users/PC/OneDrive/Desktop/chromedriver-win64/chromedriver.exe')`:**

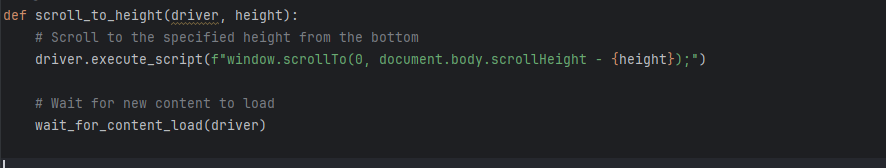
- Creates an instance of the `ChromeService` class, specifying the path to the ChromeDriver executable. The ChromeDriver is a separate executable that WebDriver uses to interact with the Chrome browser.

**8. `driver = webdriver.Chrome(service=chrome\_service, options=chrome\_options)`:**

- Creates an instance of the Chrome WebDriver (`webdriver.Chrome`), passing the `chrome\_service` and `chrome\_options` as parameters. This initializes the Chrome browser with the specified options and service.

**9. `return driver`:**

- Returns the initialized WebDriver instance.



**1. `def scroll\_to\_height(driver, height):`:**

- This line defines a function named `scroll\_to\_height` that takes two parameters:

- `driver`: The Selenium WebDriver instance, which is used to control the web browser.

- `height`: The specified height from the bottom to which the page should be scrolled.

**2. `driver.execute\_script(f"window.scrollTo(0, document.body.scrollHeight - {height});")`:**

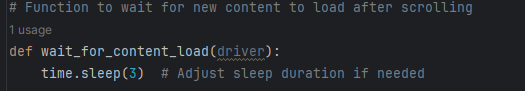
- Uses the `execute\_script` method of the Selenium WebDriver to execute a JavaScript script. The script scrolls the web page to the specified height from the bottom.

- `window.scrollTo(0, document.body.scrollHeight - {height})` is the JavaScript code that instructs the browser to scroll to the specified height.

**3. `wait\_for\_content\_load(driver)`:**

- Calls the `wait\_for\_content\_load` function (presumably defined elsewhere in the script) to wait for new content to load after scrolling.

- The purpose of waiting is to ensure that the content has loaded completely before further actions are taken.



**1. `def wait\_for\_content\_load(driver):`:**

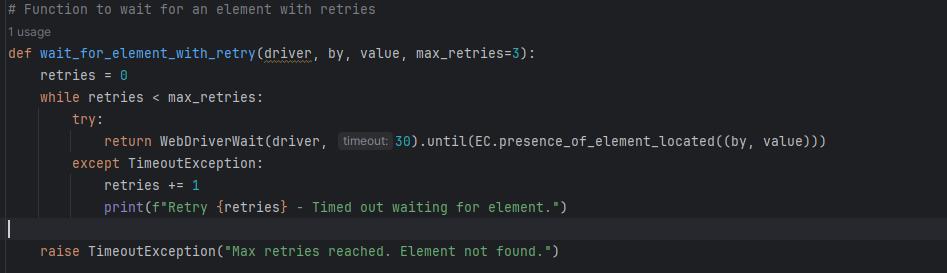
- This line defines a function named `wait\_for\_content\_load` that takes a single parameter:

- `driver`: The Selenium WebDriver instance, which is used to control the web browser.

**2. `time.sleep(3)`:**

- Uses the `sleep` function from the `time` module to pause the script for 3 seconds.

- The comment indicates that the sleep duration can be adjusted if needed.



**1. `def wait\_for\_element\_with\_retry(driver, by, value, max\_retries=3):`:**

- This line defines a function named `wait\_for\_element\_with\_retry` that takes four parameters:

- `driver`: The Selenium WebDriver instance, which is used to control the web browser.

- `by`: The method used to locate the element (e.g., By.ID, By.CLASS\_NAME).

- `value`: The value of the element's locator (e.g., the ID or class name).

- `max\_retries`: The maximum number of retry attempts, with a default value of 3.

**2. `retries = 0`:**

- Initializes a variable `retries` to 0. This variable will be used to count the number of retry attempts.

**3. `while retries < max\_retries:`:**

- Starts a `while` loop that continues until the number of retries exceeds the maximum allowed retries.

**4. `try:`:**

- Begins a `try` block where the code attempts to find the specified element using `WebDriverWait`.

**5. `return WebDriverWait(driver, 30).until(EC.presence\_of\_element\_located((by, value)))`:**

- Uses `WebDriverWait` to wait for a maximum of 30 seconds until the specified element (identified by `by` and `value`) is present on the page.

- If the element is found within the specified time, the function returns the located element.

**6. `except TimeoutException:`:**

- Catches a `TimeoutException` if the element is not found within the specified time.

**7. `retries += 1`:**

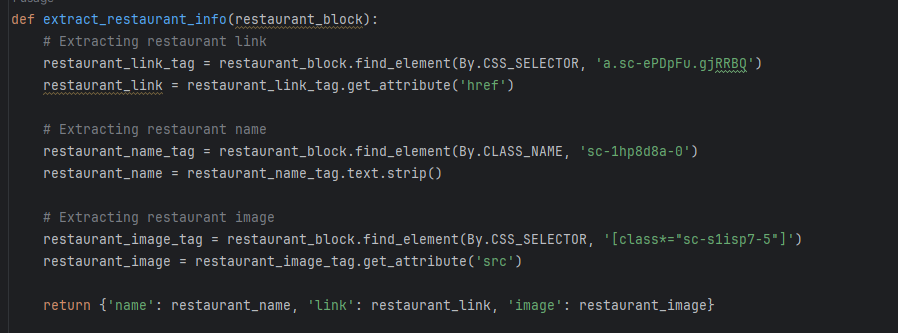
- Increments the `retries` counter.

**8. `print(f"Retry {retries} - Timed out waiting for element.")`:**

- Prints a message indicating the retry attempt and that the element timed out.

**9. `raise TimeoutException("Max retries reached. Element not found.")`:**

- If the maximum number of retries is reached and the element is still not found, the function raises a `TimeoutException` with an appropriate message.



**1. `def extract\_restaurant\_info(restaurant\_block):`:**

- This line defines a function named `extract\_restaurant\_info` that takes one parameter:

- `restaurant\_block`: A BeautifulSoup element representing a block of HTML containing information about a restaurant.

**2. `restaurant\_link\_tag = restaurant\_block.find\_element(By.CSS\_SELECTOR, 'a.sc-ePDpFu.gjRRBQ')`:**

- Uses the `find\_element` method to locate an anchor (`a`) element within the `restaurant\_block` using a CSS selector.

- The CSS selector is `'a.sc-ePDpFu.gjRRBQ'`, indicating an anchor element with the specified classes.

- Assigns the found element to the variable `restaurant\_link\_tag`.

**3. `restaurant\_link = restaurant\_link\_tag.get\_attribute('href')`:**

- Gets the value of the 'href' attribute from the `restaurant\_link\_tag`, representing the link to the restaurant.

- Assigns the extracted link to the variable `restaurant\_link`.

**4. `restaurant\_name\_tag = restaurant\_block.find\_element(By.CLASS\_NAME, 'sc-1hp8d8a-0')`:**

- Uses the `find\_element` method to locate an element with a specified class name within the `restaurant\_block`.

- Assigns the found element to the variable `restaurant\_name\_tag`.

**5. `restaurant\_name = restaurant\_name\_tag.text.strip()`:**

- Gets the text content from the `restaurant\_name\_tag` and strips leading and trailing whitespaces.

- Assigns the cleaned text to the variable `restaurant\_name`.

**6. `restaurant\_image\_tag = restaurant\_block.find\_element(By.CSS\_SELECTOR, '[class\*="sc-s1isp7-5"]')`:**

- Uses the `find\_element` method to locate an element within the `restaurant\_block` using a CSS selector.

- The selector targets an element with a class containing the substring 'sc-s1isp7-5'.

- Assigns the found element to the variable `restaurant\_image\_tag`.

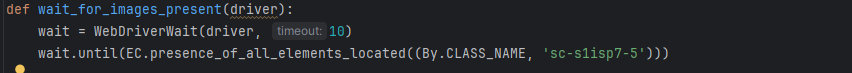
**7. `restaurant\_image = restaurant\_image\_tag.get\_attribute('src')`:**

- Gets the value of the 'src' attribute from the `restaurant\_image\_tag`, representing the link to the restaurant image.

- Assigns the extracted image link to the variable `restaurant\_image`.

**8. `return {'name': restaurant\_name, 'link': restaurant\_link, 'image': restaurant\_image}`:**

- Returns a dictionary containing the extracted information: restaurant name, link, and image.



**1. `def wait\_for\_images\_present(driver):`:**

- This line defines a function named `wait\_for\_images\_present` that takes a single parameter:

- `driver`: The Selenium WebDriver instance, which is used to control the web browser.

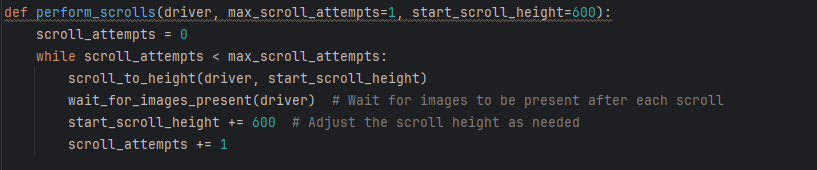
**2. `wait = WebDriverWait(driver, 10)`:**

- Creates a `WebDriverWait` instance named `wait` with a maximum timeout of 10 seconds. This means that the script will wait for up to 10 seconds for a certain condition to be met.

**3. `wait.until(EC.presence\_of\_all\_elements\_located((By.CLASS\_NAME, 'sc-s1isp7-5')))`:**

- Uses the `until` method of the `wait` instance to wait until a condition is met. The condition, specified using the `EC.presence\_of\_all\_elements\_located` expected condition, waits for all elements with a specific class name to be present on the page.

- The class name used in this case is 'sc-s1isp7-5', indicating that it's waiting for elements with this class to be present.



**1. `def perform\_scrolls(driver, max\_scroll\_attempts=1, start\_scroll\_height=600):`:**

- This line defines a function named `perform\_scrolls` that takes three parameters:

- `driver`: The Selenium WebDriver instance, which is used to control the web browser.

- `max\_scroll\_attempts`: The maximum number of scroll attempts (default is 1).

- `start\_scroll\_height`: The initial scroll height from the bottom of the page (default is 600).

**2. `scroll\_attempts = 0`:**

- Initializes a variable `scroll\_attempts` to 0. This variable will be used to count the number of scroll attempts.

**3. `while scroll\_attempts < max\_scroll\_attempts:`:**

- Starts a `while` loop that continues until the number of scroll attempts exceeds the maximum allowed scroll attempts.

**4. `scroll\_to\_height(driver, start\_scroll\_height)`:**

- Calls the `scroll\_to\_height` function (presumably defined elsewhere in the script) to scroll the page to the specified height from the bottom.

- The `start\_scroll\_height` is initially set to 600 and is adjusted after each scroll.

**5. `wait\_for\_images\_present(driver)`:**

- Calls the `wait\_for\_images\_present` function (presumably defined elsewhere in the script) to wait for images to be present on the page after each scroll.

- This ensures that the script waits until the new content, including images, has loaded.

**6. `start\_scroll\_height += 600`:**

- Increases the `start\_scroll\_height` by 600 after each scroll. This determines the next scroll position.

**7. `scroll\_attempts += 1`:**

- Increments the `scroll\_attempts` counter.



**`driver = initialize\_driver(headless=False)`:**

- Calls the `initialize\_driver` function, passing the argument `headless=False`.

- The `initialize\_driver` function, presumably defined elsewhere in the script, initializes and configures the Chrome WebDriver based on the specified options.

- In this specific case, `headless=False` means that the Chrome browser will not run in headless mode, and a graphical user interface will be visible during the automation process.

- The result is assigned to the variable `driver`, which is then used to control the Chrome browser throughout the script.



**1. `perform\_scrolls(driver, max\_scroll\_attempts=42, start\_scroll\_height=2000)`:**

- Calls the function `perform\_scrolls` with three arguments:

- `driver`: The Selenium WebDriver instance, which is used to control the web browser.

- `max\_scroll\_attempts`: The maximum number of scroll attempts, set to 42 by default.

- `start\_scroll\_height`: The initial scroll height from the bottom of the page, set to 2000 by default.

**2. `perform\_scrolls`:**

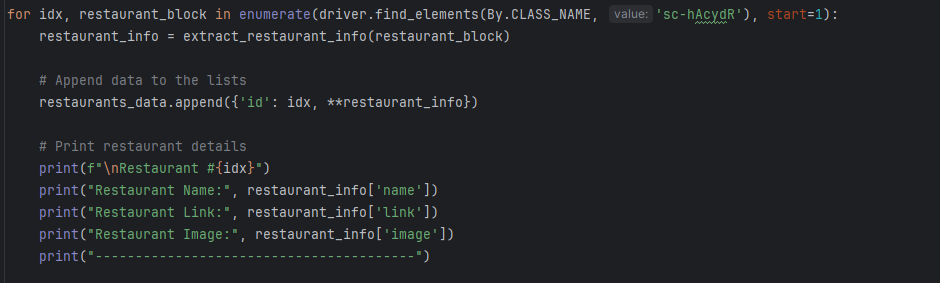
- This is the previously defined function (presumably elsewhere in the script) that performs multiple scrolls on a web page, waiting for images to be present after each scroll. It takes the specified parameters to control the scrolling behavior.

**3. `max\_scroll\_attempts=42`:**

- The maximum number of scroll attempts is set to 42. This means that the web page will be scrolled 42 times in total.

**4. `start\_scroll\_height=2000`:**

- The initial scroll height from the bottom of the page is set to 2000 pixels. This determines the starting position for the first scroll.



**1. `for idx, restaurant\_block in enumerate(driver.find\_elements(By.CLASS\_NAME, 'sc-hAcydR'), start=1):`:**

- Initiates a loop to iterate over elements found on the web page with the class name 'sc-hAcydR'.

- `idx` is an index variable starting from 1, and `restaurant\_block` represents each individual block containing information about a restaurant.

**2. `restaurant\_info = extract\_restaurant\_info(restaurant\_block)`:**

- Calls the `extract\_restaurant\_info` function to extract information from the current `restaurant\_block`.

- The extracted information is stored in the `restaurant\_info` variable.

**3. `restaurants\_data.append({'id': idx, restaurant\_info})`:**

- Appends a dictionary to the `restaurants\_data` list. The dictionary includes the index (`id`), and it also unpacks the content of `restaurant\_info` using the `` operator, effectively adding all key-value pairs from `restaurant\_info` to the dictionary.

**4. `print(f"\nRestaurant #{idx}")`:**

- Prints the restaurant index.

**5. `print("Restaurant Name:", restaurant\_info['name'])`:**

- Prints the name of the restaurant.

**6. `print("Restaurant Link:", restaurant\_info['link'])`:**

- Prints the link to the restaurant.

**7. `print("Restaurant Image:", restaurant\_info['image'])`:**

- Prints the link to the restaurant image.

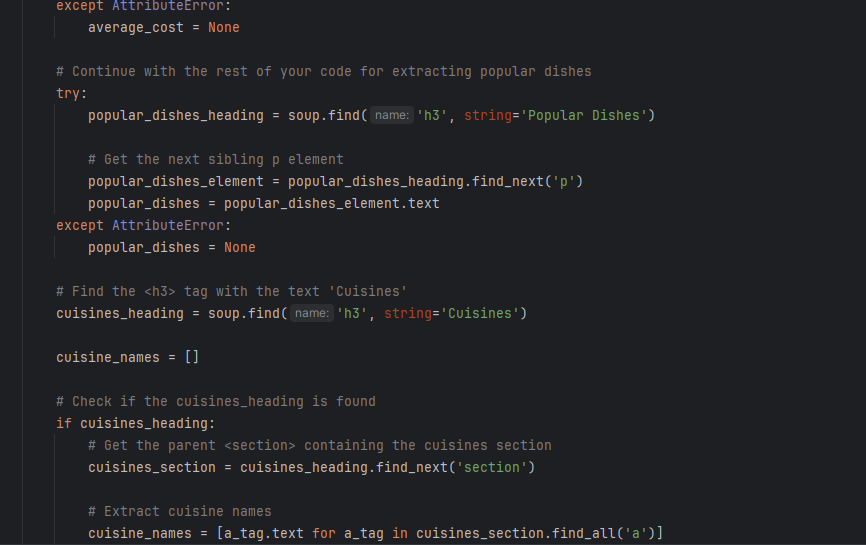
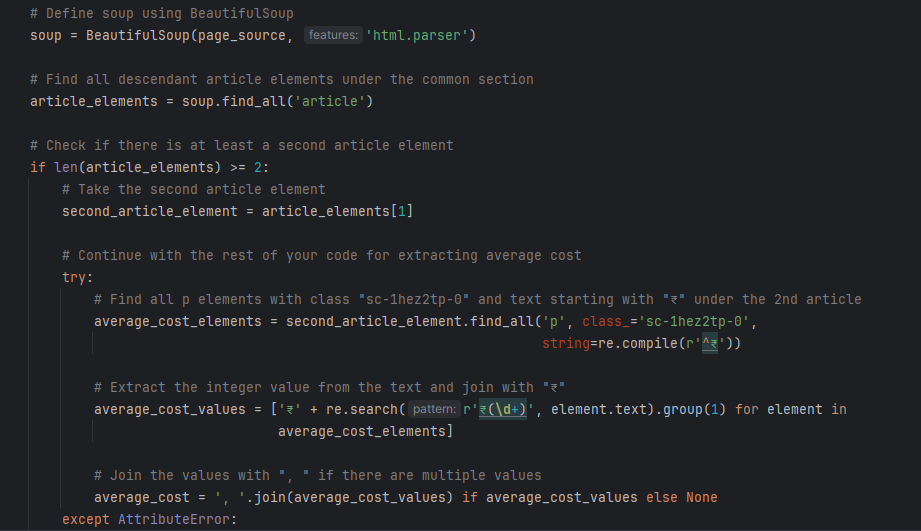
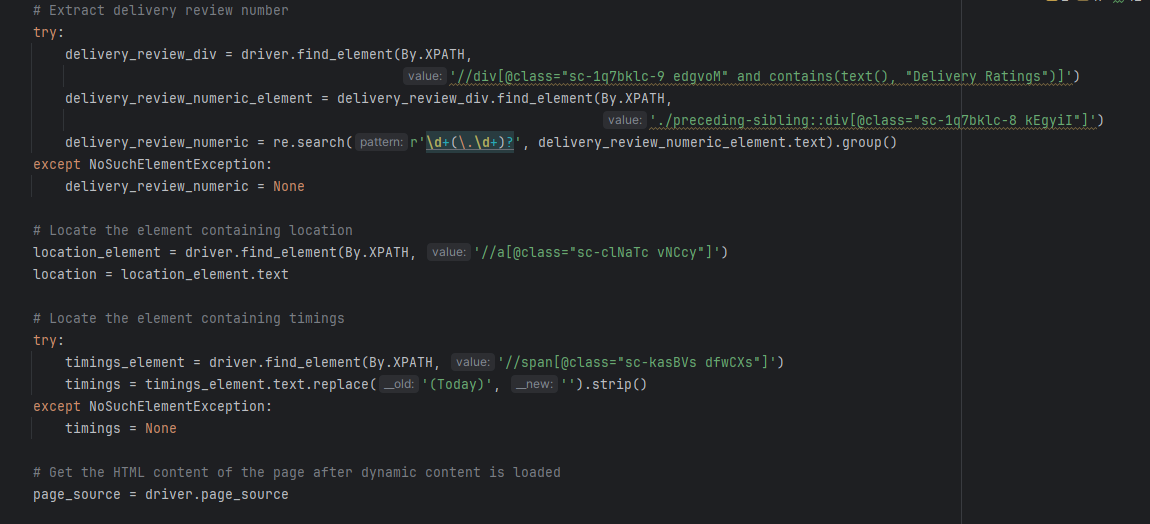
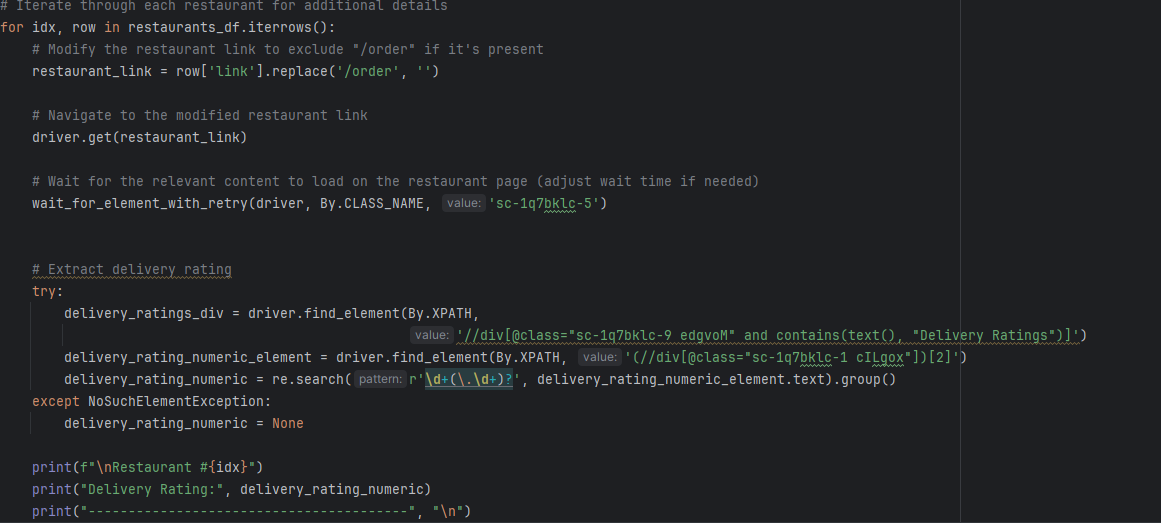
**8. `print("----------------------------------------")`:**

- Prints a separator line for better readability.

**9. `restaurants\_df = pd.DataFrame(restaurants\_data)`:**

- After the loop, a pandas DataFrame (`restaurants\_df`) is created using the list of dictionaries (`restaurants\_data`).

- This DataFrame will store the extracted information for each restaurant, making it easier to analyze and manipulate the data.



**1. `for idx, row in restaurants\_df.iterrows():`:**

- Iterates over each row in the DataFrame (`restaurants\_df`). `idx` is the index of the row, and `row` represents the data in that row.

**2. `restaurant\_link = row['link'].replace('/order', '')`:**

- Modifies the restaurant link by removing the '/order' part if it's present.

**3. `driver.get(restaurant\_link)`:**

- Navigates to the modified restaurant link using the Selenium WebDriver.

**4. `wait\_for\_element\_with\_retry(driver, By.CLASS\_NAME, 'sc-1q7bklc-5')`:**

- Waits for an element with the class name 'sc-1q7bklc-5' to be present on the page.

**5. `try: ... except NoSuchElementException: ...`:**

- Uses a try-except block to handle exceptions, specifically `NoSuchElementException`.

- Inside the try block, it extracts the delivery rating and delivery review number using XPath expressions.

**6. `location\_element = driver.find\_element(By.XPATH, '//a[@class="sc-clNaTc vNCcy"]')`:**

- Locates an element with the XPath expression to find the location.

**7. `try: ... except NoSuchElementException: ...`:**

- Similar to the previous try-except block, it handles exceptions to extract timings using an XPath expression.

**8. `page\_source = driver.page\_source`:**

- Gets the HTML content of the page after dynamic content is loaded.

**9. `soup = BeautifulSoup(page\_source, 'html.parser')`:**

- Creates a BeautifulSoup object using the HTML content.

**10. `article\_elements = soup.find\_all('article')`:**

- Finds all descendant article elements under the common section in the soup.

**11. `if len(article\_elements) >= 2:`:**

- Checks if there is at least a second article element.

**12. `second\_article\_element = article\_elements[1]`:**

- If there is a second article element, it selects and continues with the code.

**13. `try: ... except AttributeError: ...`:**

- Tries to extract the average cost from the second article element, handling AttributeError exceptions.

**14. `try: ... except AttributeError: ...`:**

- Tries to extract popular dishes from the HTML, handling AttributeError exceptions.

**15. `cuisines\_heading = soup.find('h3', string='Cuisines')`:**

- Finds the `<h3>` tag with the text 'Cuisines' in the soup.

**16. `if cuisines\_heading:`:**

- Checks if the cuisines heading is found.

**17. `cuisines\_section = cuisines\_heading.find\_next('section')`:**

- Finds the parent `<section>` containing the cuisines section.

**18. `known\_for\_heading = soup.find('h3', string='People Say This Place Is Known For')`:**

- Finds the `<h3>` tag with the text 'People Say This Place Is Known For' in the soup.

**19. `if known\_for\_heading:`:**

- Checks if the known for heading is found.

**20. `known\_for\_element = known\_for\_heading.find\_next('p')`:**

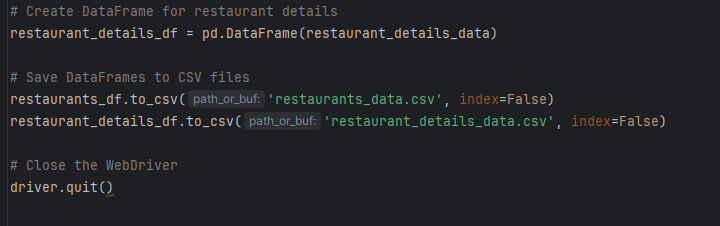
- Finds the next sibling `<p>` element after the known for heading.

**21. `restaurant\_details\_data.append({...})`:**

- Appends a dictionary with extracted details to the `restaurant\_details\_data` list.

**22. `restaurant\_details\_df = pd.DataFrame(restaurant\_details\_data)`:**

- Creates a new DataFrame (`restaurant\_details\_df`) using the list of dictionaries (`restaurant\_details\_data`).



**1. `restaurants\_df.to\_csv('restaurants\_data.csv', index=False)`:**

- This line exports the `restaurants\_df` DataFrame to a CSV file named 'restaurants\_data.csv'.

- The parameter `index=False` specifies that the index column should not be included in the CSV file.

**2. `restaurant\_details\_df.to\_csv('restaurant\_details\_data.csv', index=False)`:**

- This line exports the `restaurant\_details\_df` DataFrame to a CSV file named 'restaurant\_details\_data.csv'.

- Similarly, `index=False` is used to exclude the index column from the CSV file.

**3. `driver.quit()`:**

- This line closes the WebDriver instance, releasing resources associated with the browser automation.