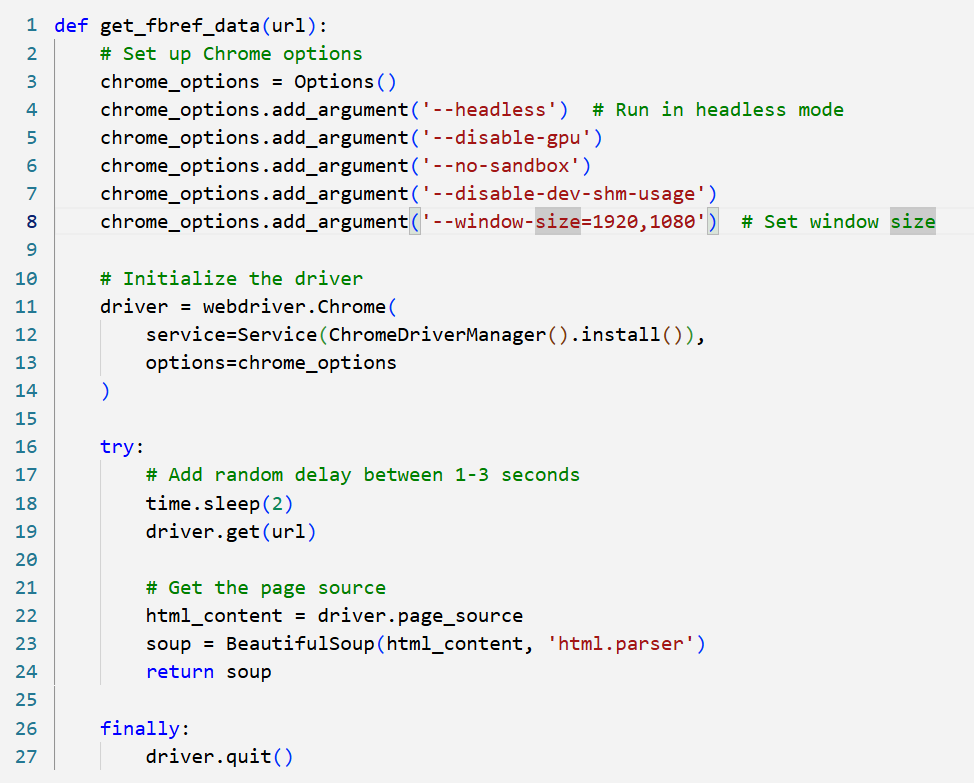
**2/. Data Collection and Description**

The data used for this project was scraped from **fbref.com**, a widely trusted source for football statistics. Specifically, the Premier League match statistics were selected due to their rich history of detailed match data, which includes various performance metrics that can be predictive of future outcomes. These statistics provide essential features such as **shots**, **possession**, **fouls**, **goals**, **venue**, and **round**, all of which were used to model the outcome of each match. The dataset spans multiple seasons, and scraping it from a reliable website ensures that the data is both comprehensive and relevant to the task.

To gather this data, web scraping techniques were employed using **Selenium** and **BeautifulSoup**. Selenium was specifically chosen for its ability to interact with dynamically loaded web pages, which is necessary because the data on fbref.com is rendered through JavaScript. The first part of the code initializes a headless browser session using Selenium and navigates to the required URL. This was done to simulate a real user interaction and retrieve the page source from which BeautifulSoup can parse the HTML content.

The get\_fbref\_data(url) function, which handles the retrieval of the page source, sets up the browser options to run in headless mode (i.e., without opening a visible window) to improve performance and efficiency. After loading the page, the content is extracted, parsed, and cleaned using BeautifulSoup. This step is essential to ensure that only the relevant match data is captured. Below is a key section of the code used to scrape the data:

The get\_fbref\_data(url) function is used to scrape match data from the **fbref.com** website by automating the browser with **Selenium** and parsing the HTML content using **BeautifulSoup**. This function handles the process of visiting the URL, rendering the webpage (including JavaScript content), and extracting the page's source code, which contains the match statistics. Below is a detailed breakdown of the function:

1. Chrome Options Setup**:**  
   The function starts by setting up **Chrome options** for the web driver. These options configure the behavior of the browser used by Selenium:
   * --headless: This option ensures that the browser runs in **headless mode**, meaning it operates without a graphical interface. This is beneficial for running the scraping script on a server or in the background without consuming unnecessary resources.
   * --disable-gpu: This disables GPU hardware acceleration, which is often recommended for headless mode to avoid potential issues.
   * --no-sandbox: Disables the sandboxing feature of Chrome, which is necessary when running Chrome in headless mode on certain systems.
   * --disable-dev-shm-usage: This option helps mitigate issues with shared memory usage, particularly in Docker containers or environments with limited resources.
   * --window-size=1920,1080: Sets the window size for the headless browser to 1920x1080 pixels, ensuring that the page is rendered properly, and all content is visible for scraping.
2. Web Driver Initialization**:**  
   After configuring the browser options, the **Selenium WebDriver** is initialized. The webdriver.Chrome() method creates a new browser instance with the specified options, and it installs the required **ChromeDriver** using ChromeDriverManager().install(). This method automatically downloads the appropriate version of ChromeDriver, which is needed to interact with the Chrome browser.
3. Page Loading and Data Extraction**:**  
   The function then navigates to the specified URL (driver.get(url)). A **random delay** of 2 seconds is introduced using time.sleep(2) to simulate real user behavior and avoid overloading the server with requests. After loading the page, the function retrieves the entire page's HTML content using driver.page\_source.
4. HTML Parsing with BeautifulSoup**:**  
   The raw HTML content is then passed to **BeautifulSoup**, which parses the HTML and creates a **BeautifulSoup object** (soup). This object provides a convenient way to navigate and extract specific data from the HTML structure. In this case, the parsed content is returned as the output of the function.
5. Driver Cleanup**:**  
   Finally, the browser instance is closed using driver.quit(), which ensures that the browser is properly terminated after the data is retrieved. This step helps release system resources and ensures that no processes are left running.

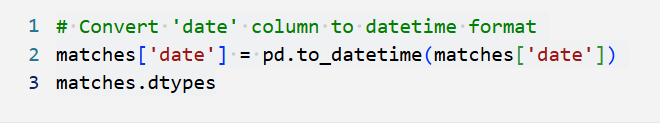
By using this function, the script can efficiently scrape data from **fbref.com**, including match statistics like shots, possession, fouls, and goals, which are essential for the machine learning model. The scraping process is carried out for multiple seasons, allowing data to be collected from various years. The data is then stored in a CSV file, which is the foundation for the modeling phase of the project.

**3/.Data Preprocessing**

Once the data was scraped, the next step was to preprocess it in order to make it suitable for modeling. The dataset was cleaned, transformed, and enriched to ensure that it was in the right format for machine learning algorithms. Several preprocessing steps were performed on the dataset, including the conversion of date and categorical columns, handling time information, and feature engineering to create new useful variables. The following outlines the key preprocessing steps and their corresponding code.

* Converting the 'date' Column to Datetime Format

The first preprocessing step was to convert the date column into a proper **datetime format**. This is an essential step, as the date column was initially stored as a string, which would not allow for efficient sorting or filtering based on time. By converting it into a datetime object, it enables time-based operations such as extracting the day of the week or calculating time differences. The conversion was done using the pd.to\_datetime() function from **Pandas**, as shown in the following code:



This line of code converts the date column into the datetime format, allowing for more efficient and flexible manipulation of date-related data.

* Encoding the 'venue' Column

The venue column, which indicates whether the match was played at home or away, is a categorical feature. Machine learning models typically do not work directly with categorical variables, so it is necessary to convert them into a numerical form. To achieve this, I used **label encoding**, which assigns a unique integer to each category. In this case, "home" and "away" matches were encoded as integer values using the astype('category').cat.codes method. Here’s the relevant code for this step:

This code converts the venue column into an integer code where each unique venue (home or away) is assigned a corresponding numeric value. This transformation is important because most machine learning algorithms require numerical inputs.

In summary, the data preprocessing steps involved converting categorical variables into numerical codes, handling time-related information, and creating new features from the existing data. These transformations ensure that the dataset is in an appropriate format for modeling. By converting date to a datetime object, encoding categorical features like venue and opponent, extracting meaningful time features such as hour and day\_code, and creating the binary target variable, the dataset is now ready to be used in machine learning algorithms.

These preprocessing steps are crucial because they prepare the data in a way that makes it understandable for machine learning models, improving the accuracy and effectiveness of the model's predictions.

**4/. Methodology: Modeling & Prediction**

After preprocessing the data, I moved on to the next phase: building a machine learning model to predict the outcome of Premier League matches. The goal of this project was to predict match outcomes (win, loss, or draw) based on match statistics such as shots, possession, fouls, goals, and venue. These match statistics are believed to hold key patterns that can be used to make informed predictions about future matches.

For this task, I chose to use a **Random Forest Classifier** model. Random Forest is an ensemble learning method that builds multiple decision trees and combines their outputs to improve prediction accuracy. It is particularly effective for handling large datasets with numerous features, as it reduces the risk of overfitting and handles non-linear relationships well. The RandomForestClassifier from **scikit-learn** was used to build the model. Here’s an important section of the code that sets up and trains the model: