Shiraz University

Linear Algebra (Spring 2024)

Project Documentation

Contributors

Matin Monshizadeh (9932122)

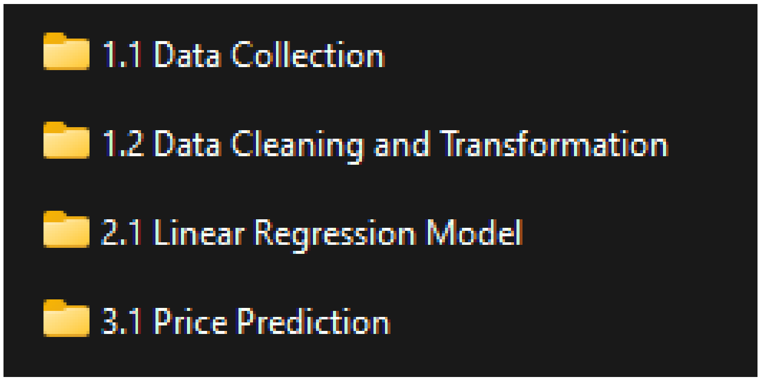
AmirHossein Roodaki (9935707)

Predict Prices With Linear Regression:

Introduction:

This project applies linear algebra to predict product prices using data extracted from e-commerce sites. Participants will gather and transform data, then train and evaluate a linear regression model. Through hands-on experience, they will enhance their understanding of data analysis and regression techniques in real-world applications.

Our project contains these steps:



1.1 Data Collection

1.2 Data Cleaning and Transformation

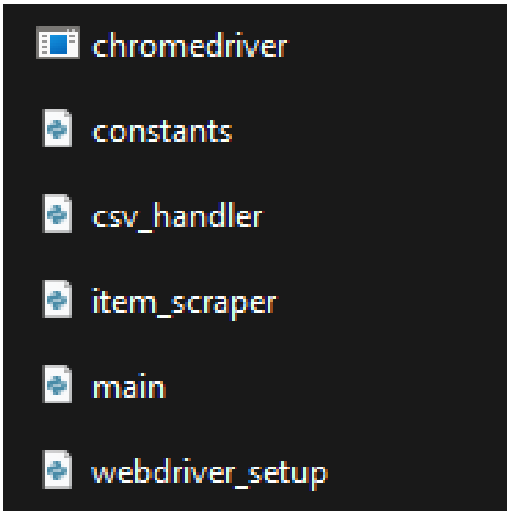
2.1 Linear Regression Model

3.1 Price Prediction

In each section, it is fully explained what processes have been followed.

1.1 Data Collection:

This folder Contains:



chromedriver

constants

csv_handler

item_scraper

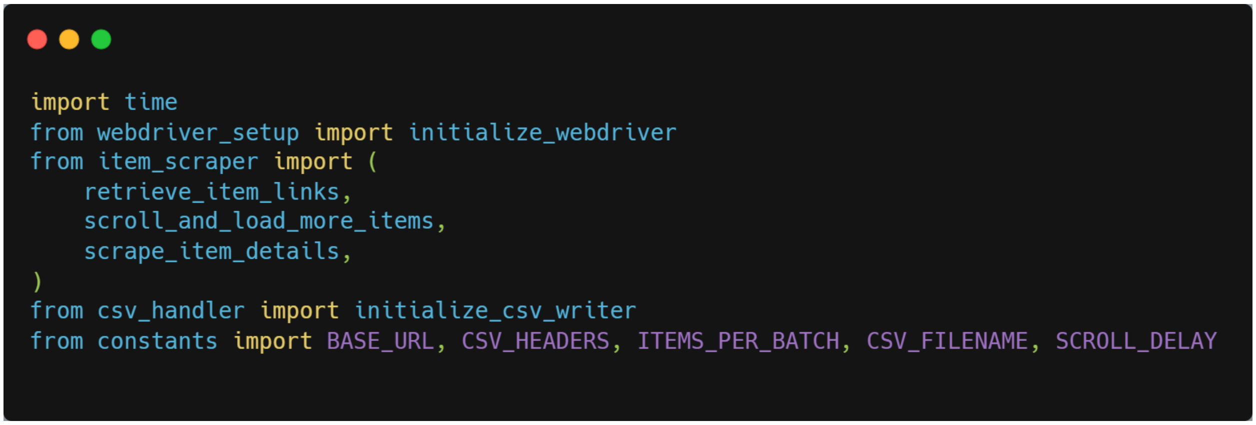
main

webdriver_setup

Main Script: main.py

This script is the entry point of the project. It orchestrates the entire web scraping process by utilizing functions from various modules to initialize the web driver, navigate the website, scrape data, and save it to a CSV file.

Import Statements

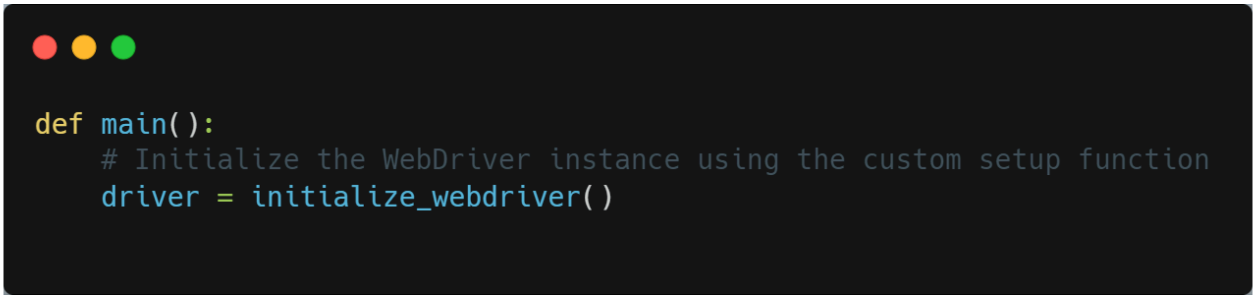


```python
import time
from webdriver_setup import initialize_webdriver
from item_scraper import (
    retrieve_item_links,
    scroll_and_load_more_items,
    scrape_item_details,
)
from csv_handler import initialize_csv_writer
from constants import BASE_URL, CSV_HEADERS, ITEMS_PER_BATCH, CSV_FILENAME, SCROLL_DELAY
```

- time: Standard Python library for adding delays.
- webdriver_setup: Module for initializing the web driver.
- item_scraper: Module containing functions for retrieving item links, scrolling and loading more items, and scraping item details.
- csv_handler: Module for handling CSV file operations.
- constants: Module containing constant values used throughout the script.

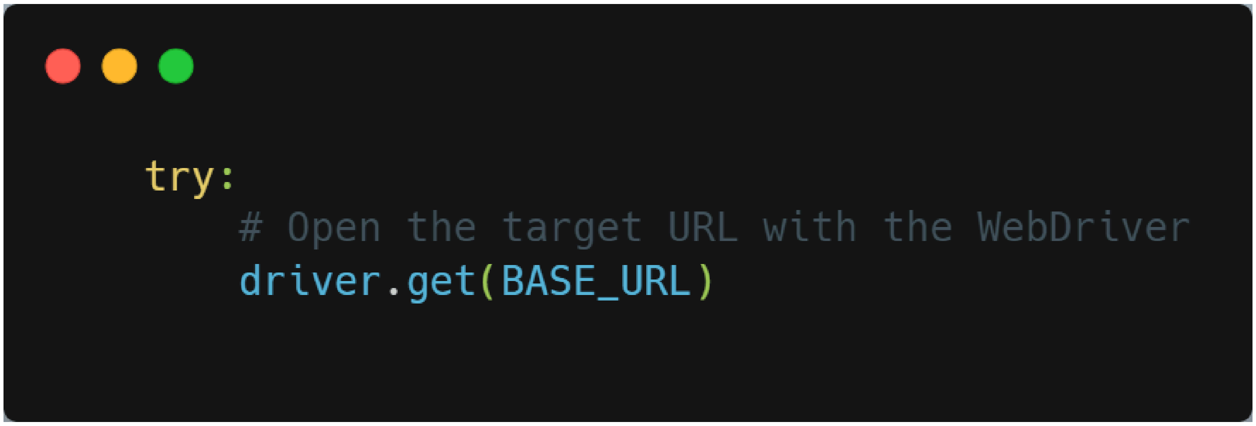
Main Function: main()

The main function is the core of the script, coordinating the entire scraping process.



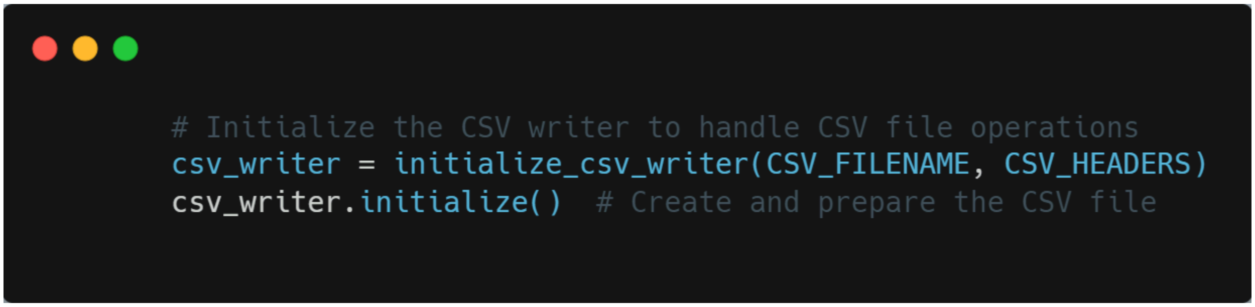
```python
def main():
    # Initialize the WebDriver instance using the custom setup function
    driver = initialize_webdriver()
```

- initialize_webdriver(): Sets up and returns a configured WebDriver instance.



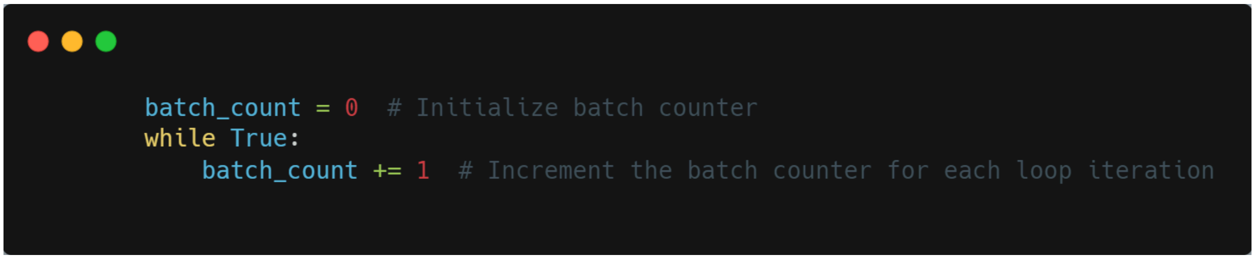
```python
try:
    # Open the target URL with the WebDriver
    driver.get(BASE_URL)
```

- driver.get(BASE_URL): Opens the Divar mobile phones section using the WebDriver.



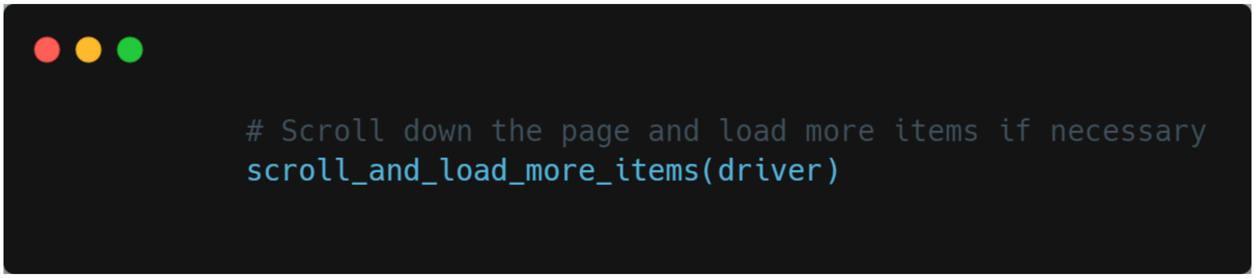
```python
    # Initialize the CSV writer to handle CSV file operations
    csv_writer = initialize_csv_writer(CSV_FILENAME, CSV_HEADERS)
    csv_writer.initialize()  # Create and prepare the CSV file
```

- initialize_csv_writer(CSV_FILENAME, CSV_HEADERS): Initializes the CSV writer with the specified filename and headers.
- csv_writer.initialize(): Creates the CSV file and writes the headers if the file does not already exist.



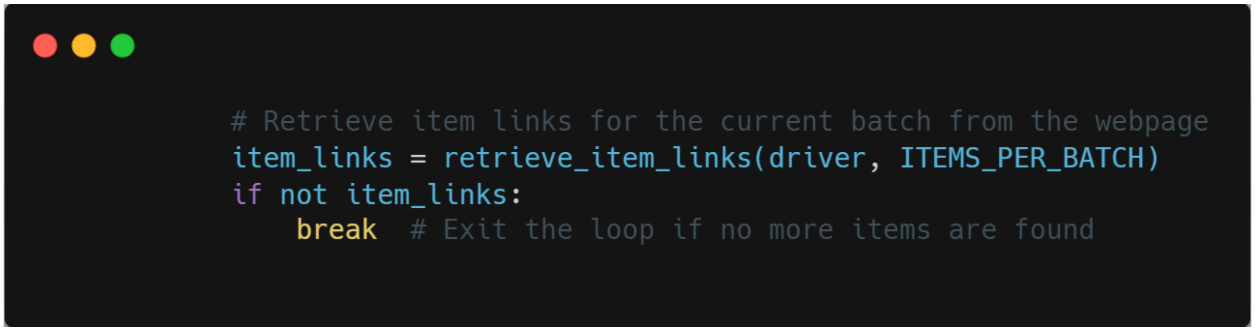
```python
    batch_count = 0  # Initialize batch counter
    while True:
        batch_count += 1  # Increment the batch counter for each loop iteration
```

- batch_count: Tracks the number of batches processed.



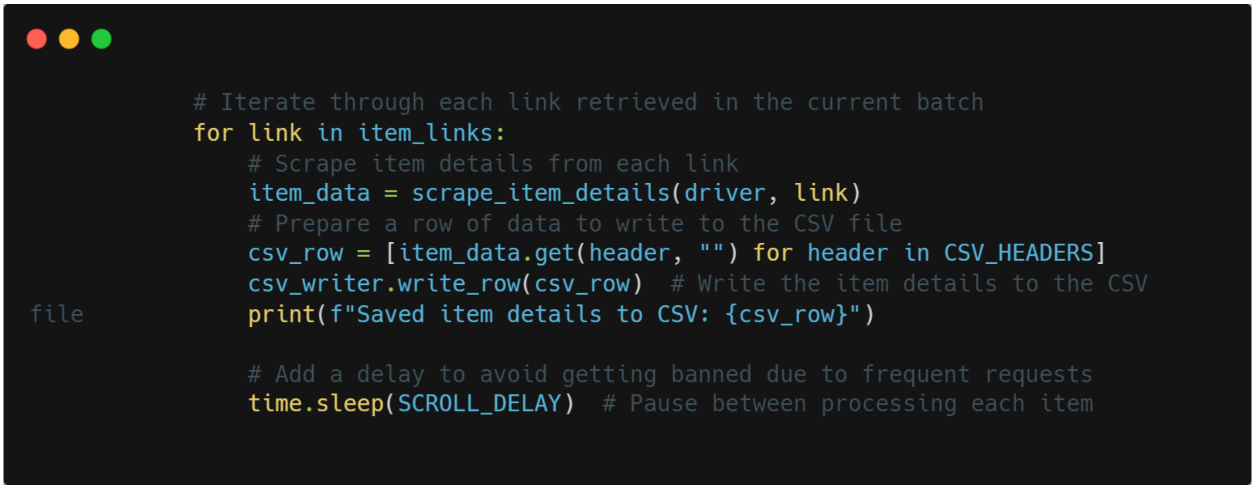
```
# Scroll down the page and load more items if necessary
scroll_and_load_more_items(driver)
```

- scroll_and_load_more_items(driver): Scrolls the webpage and clicks the 'Load more Items' button if present.



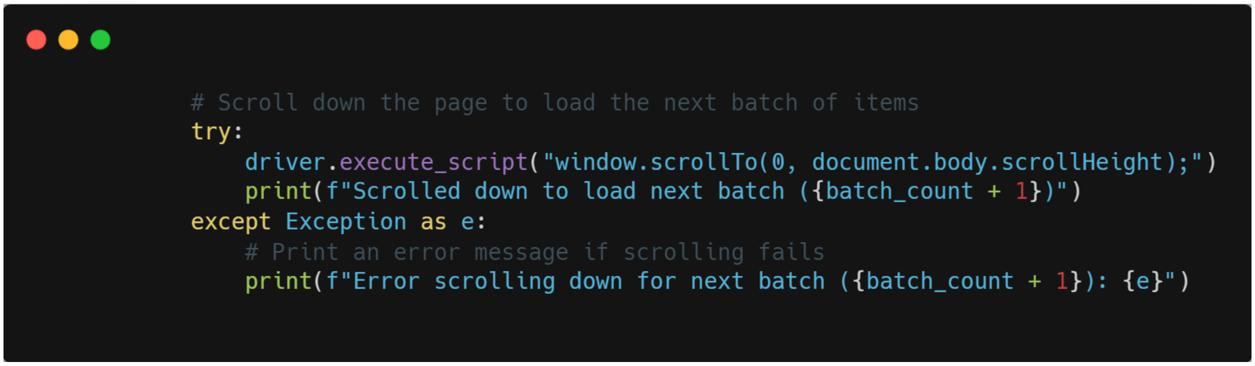
```
# Retrieve item links for the current batch from the webpage
item_links = retrieve_item_links(driver, ITEMS_PER_BATCH)
if not item_links:
    break  # Exit the loop if no more items are found
```

- retrieve_item_links(driver, ITEMS_PER_BATCH): Retrieves a batch of item links from the page. Exits the loop if no more items are found.



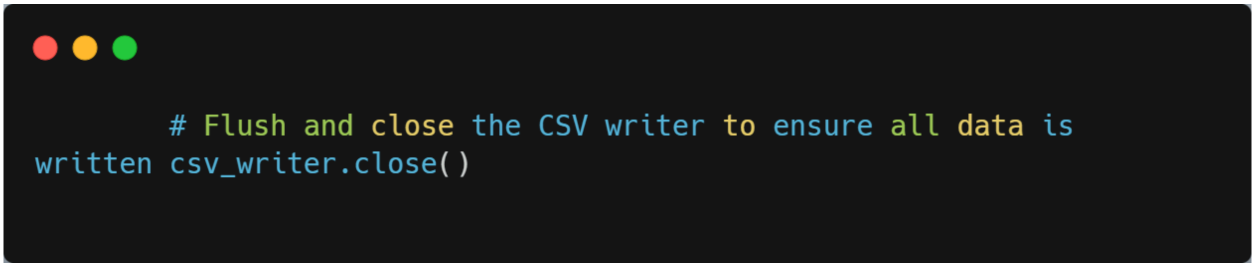
```
      # Iterate through each link retrieved in the current batch
      for link in item_links:
        # Scrape item details from each link
        item_data = scrape_item_details(driver, link)
        # Prepare a row of data to write to the CSV file
        csv_row = [item_data.get(header, "") for header in CSV_HEADERS]
        csv_writer.write_row(csv_row)  # Write the item details to the CSV
file    print(f"Saved item details to CSV: {csv_row}")

        # Add a delay to avoid getting banned due to frequent requests
        time.sleep(SCROLL_DELAY)  # Pause between processing each item
```

- scrape_item_details(driver, link): Scrapes details of an item from its detail page.
- csv_writer.write_row(csv_row): Writes the scraped item details to the CSV file.
- time.sleep(SCROLL_DELAY): Adds a delay to avoid overloading the server and to prevent getting banned.



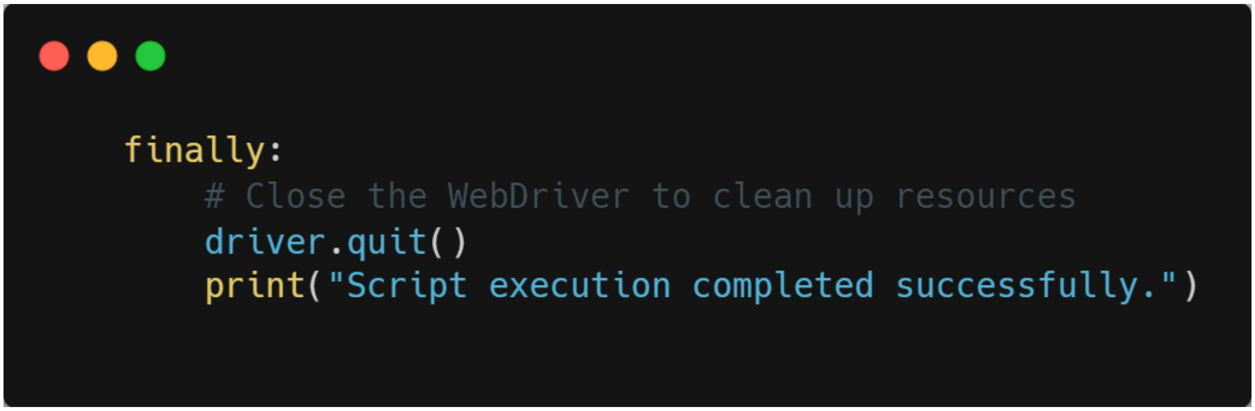
```
# Scroll down the page to load the next batch of items
try:
    driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")
    print(f"Scrolled down to load next batch ({batch_count + 1})")
except Exception as e:
    # Print an error message if scrolling fails
    print(f"Error scrolling down for next batch ({batch_count + 1}): {e}")
```

- driver.execute_script("window.scrollTo(0, document.body.scrollHeight);"): Scrolls to the bottom of the page to load more items.
- Exception Handling: Catches and logs any errors that occur during scrolling.



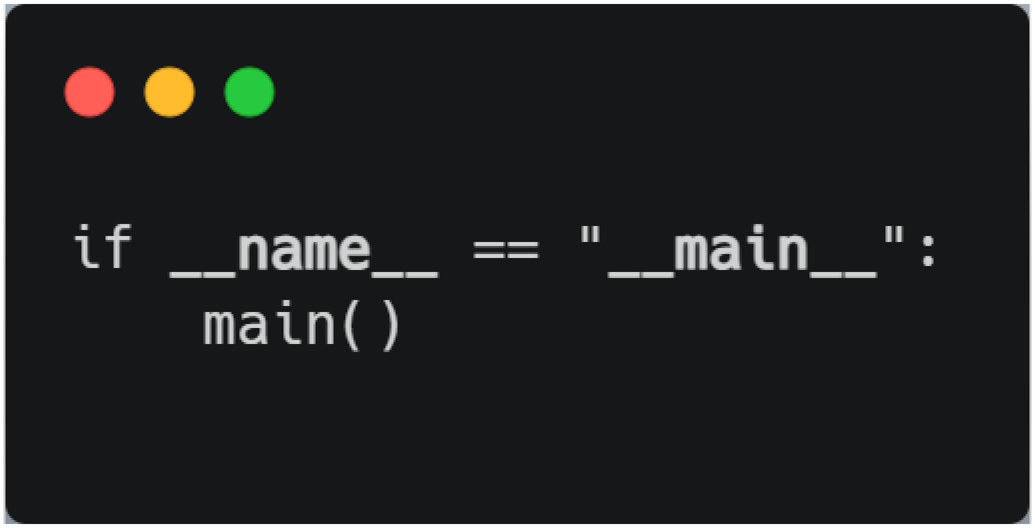
```
        # Flush and close the CSV writer to ensure all data is
written csv_writer.close()
```

- csv_writer.close(): Ensures all buffered data is written to the CSV file and closes the file.



```
finally:
    # Close the WebDriver to clean up resources
    driver.quit()
    print("Script execution completed successfully.")
```

- driver.quit(): Closes the WebDriver and cleans up resources.
- Logging: Prints a message indicating the successful completion of the script.



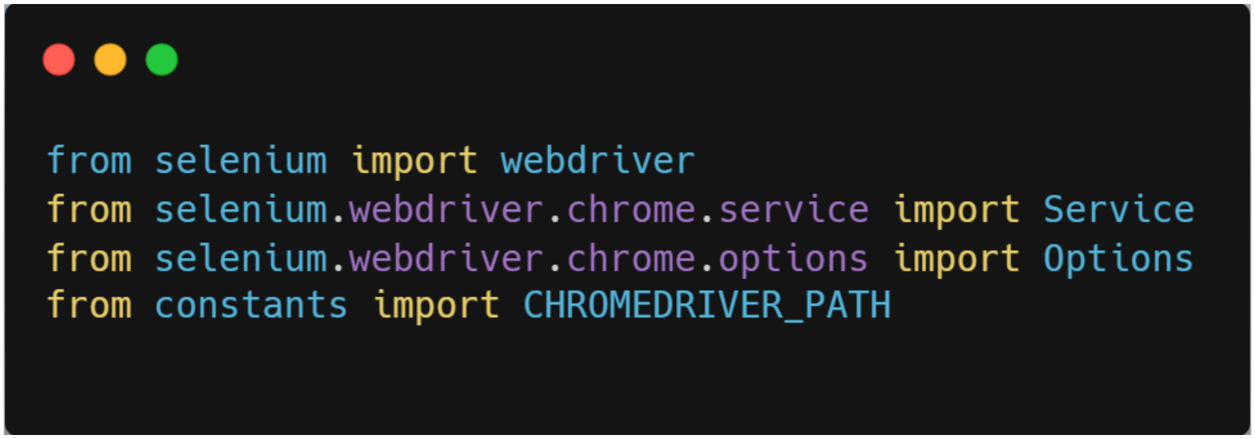
```python
if __name__ == "__main__":
    main()
```

- Entry Point: Ensures the main function is called when the script is executed directly.

Module Documentation: webdriver_setup.py

This module is responsible for setting up and configuring the Selenium WebDriver, which is used to automate browser interactions. Specifically, it configures the WebDriver to use the Chrome browser in a maximized window and incognito mode.

Import Statements

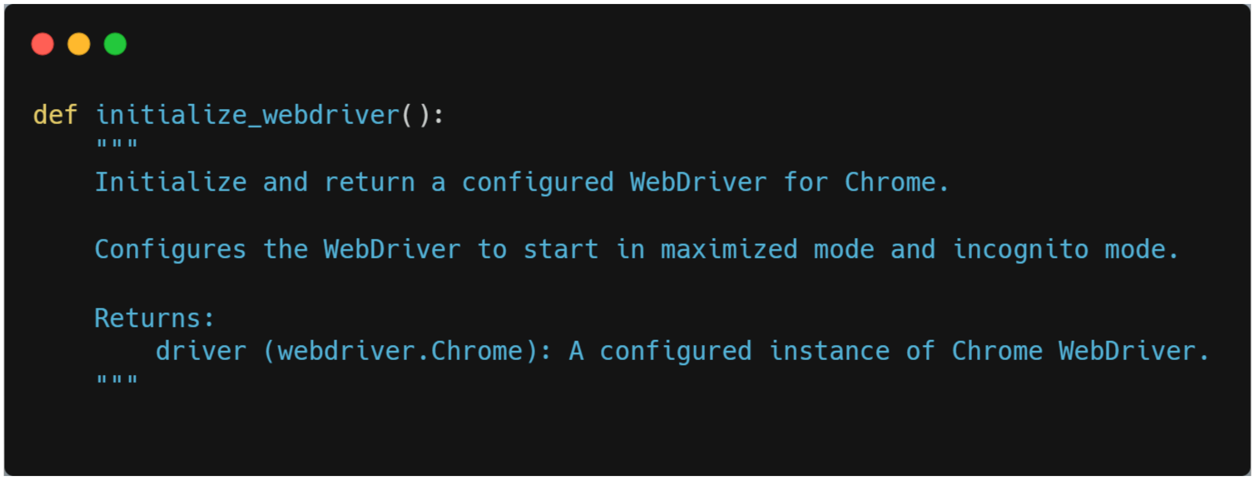


```python
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options
from constants import CHROMEDRIVER_PATH
```

- selenium.webdriver: The main Selenium WebDriver module for browser automation.
- selenium.webdriver.chrome.service.Service: Allows specifying the path to the ChromeDriver executable.
- selenium.webdriver.chrome.options.Options: Used to set various options for the Chrome browser.
- constants.CHROMEDRIVER_PATH: Imports the path to the ChromeDriver executable from the constants module.

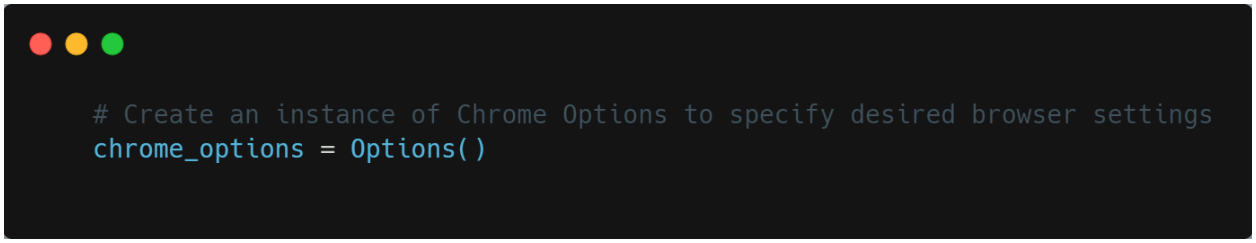
Function: initialize_webdriver()

The initialize_webdriver function sets up and returns a configured instance of the Chrome WebDriver.



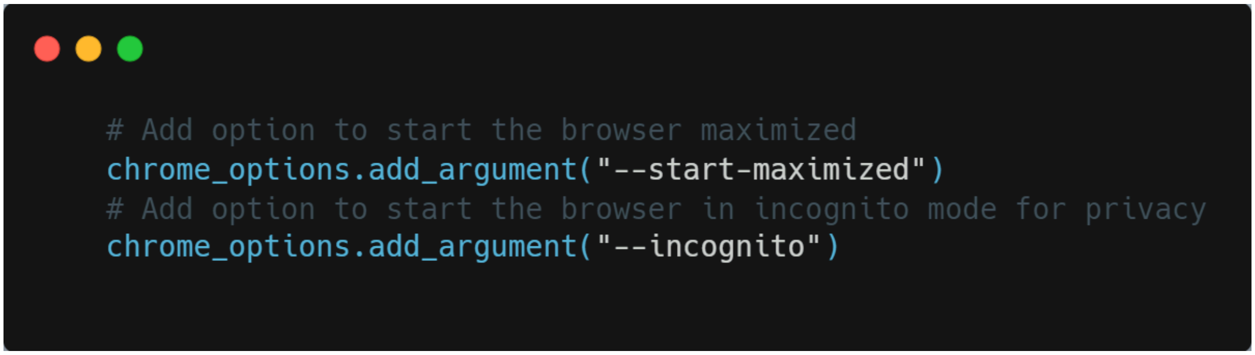
```python
def initialize_webdriver():
    """
    Initialize and return a configured WebDriver for Chrome.

    Configures the WebDriver to start in maximized mode and incognito mode.

    Returns:
        driver (webdriver.Chrome): A configured instance of Chrome WebDriver.
    """
```

- Docstring: Provides a brief description of the function, its configuration settings, and the return value.



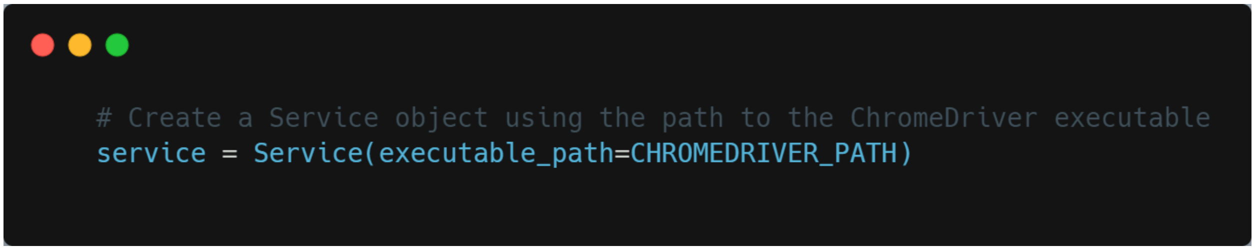
```python
    # Create an instance of Chrome Options to specify desired browser settings
    chrome_options = Options()
```

- Options(): Creates an instance of Chrome Options to specify custom browser settings.



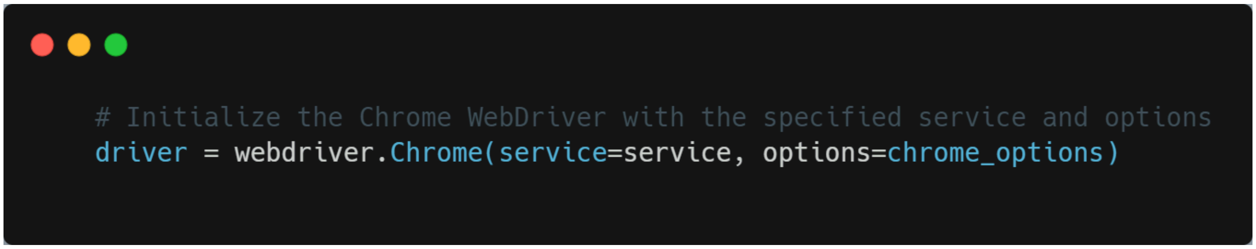
```python
    # Add option to start the browser maximized
    chrome_options.add_argument("--start-maximized")
    # Add option to start the browser in incognito mode for privacy
    chrome_options.add_argument("--incognito")
```

- chrome_options.add_argument("--start-maximized"): Configures the browser to start maximized.
- chrome_options.add_argument("--incognito"): Configures the browser to start in incognito mode for privacy.



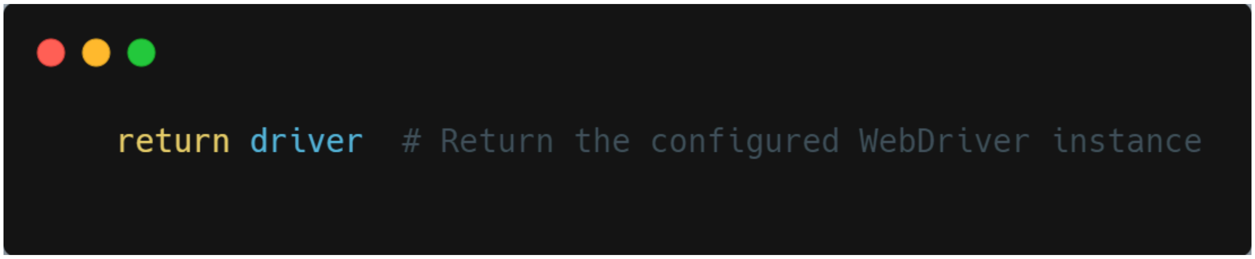
```
# Create a Service object using the path to the ChromeDriver executable
service = Service(executable_path=CHROMEDRIVER_PATH)
```

- Service(executable_path=CHROMEDRIVER_PATH): Creates a Service object with the path to the ChromeDriver executable. This specifies which ChromeDriver to use.



```
# Initialize the Chrome WebDriver with the specified service and options
driver = webdriver.Chrome(service=service, options=chrome_options)
```

- webdriver.Chrome(service=service, options=chrome_options): Initializes the Chrome WebDriver with the specified service and options.



```
return driver  # Return the configured WebDriver instance
```

- Return Value: Returns the configured WebDriver instance for use in other parts of the script.

Explanation of Key Components

1. WebDriver Configuration:
   - The function sets up the Chrome WebDriver to run with specific options: maximized window and incognito mode. This ensures that the browser window is fully visible and that browsing data (like cookies) is not stored, which can be useful for repeated testing and scraping.
2. Service Object:

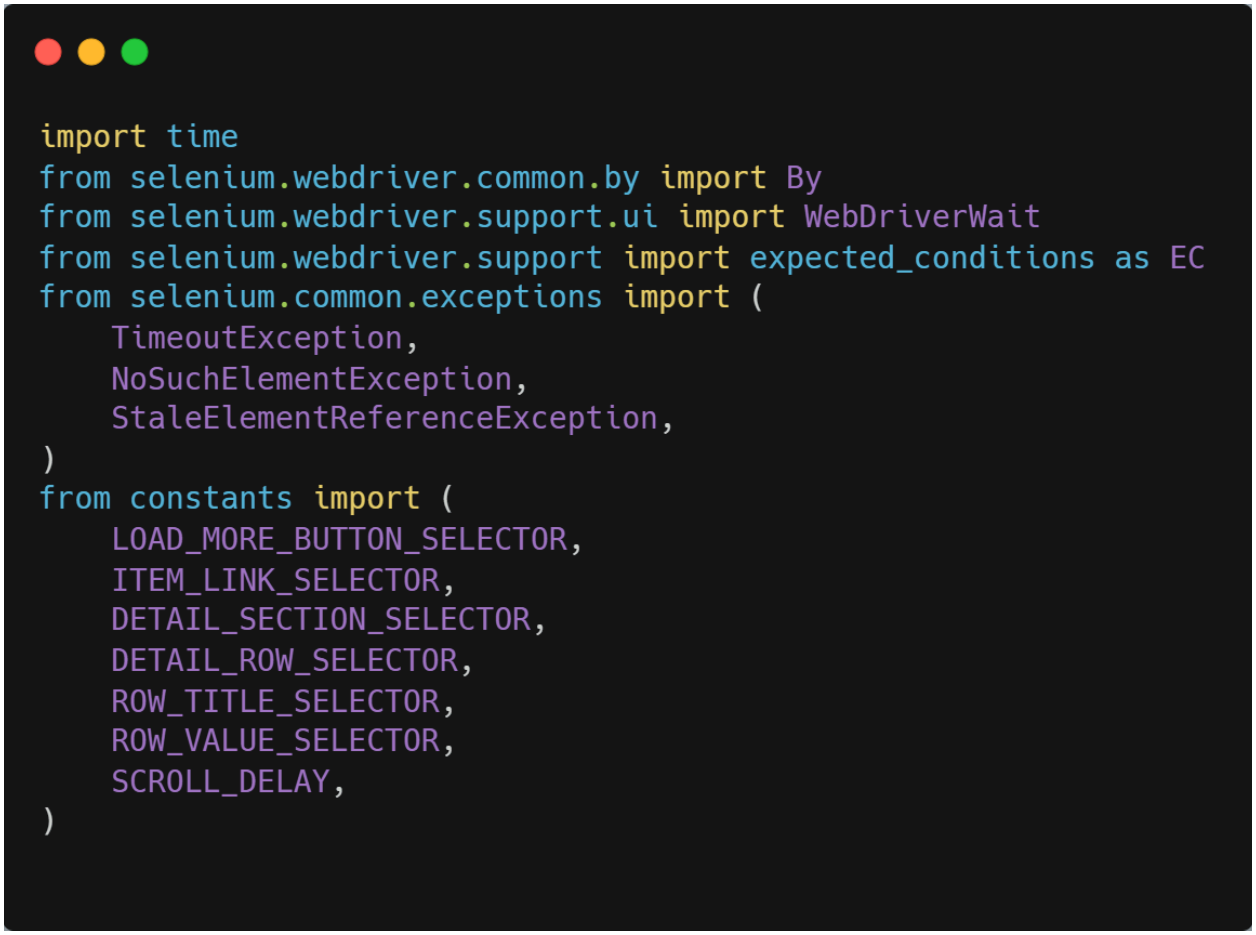
    - The Service object is created with the path to the ChromeDriver executable. This ensures that the WebDriver knows where to find the ChromeDriver, which is necessary for controlling the Chrome browser.

3. Options Object:
    - The Options object allows for various customizations of the browser’s behavior. In this case, starting maximized and in incognito mode are the chosen settings.

Module Documentation: item_scraper.py

This module is responsible for scraping item details from a webpage using Selenium WebDriver. It includes functions to retrieve item links, scroll and load more items, and scrape detailed information from item pages.

Import Statements



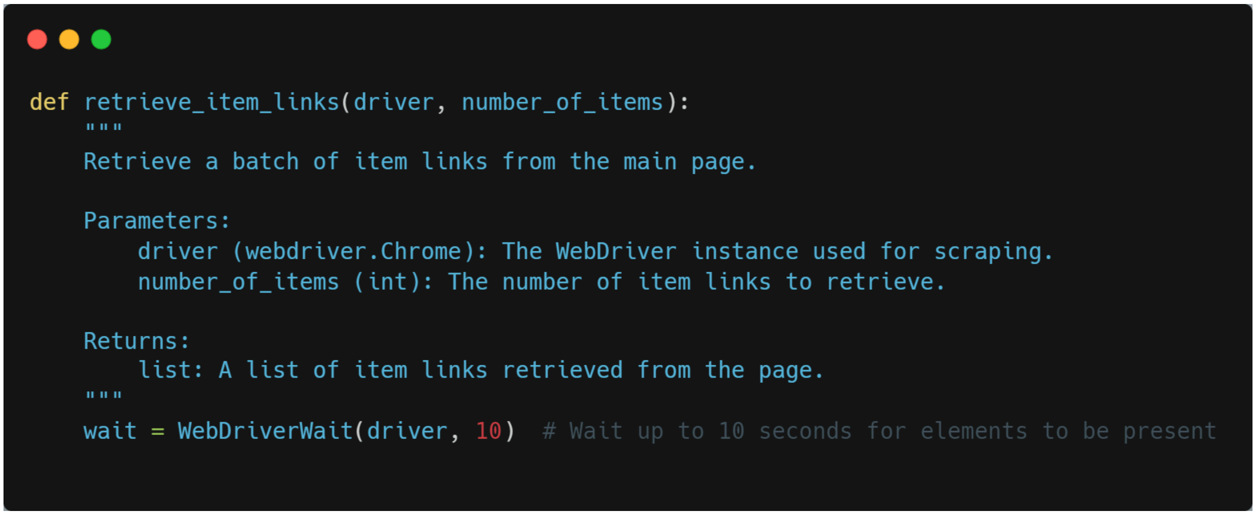
```python
import time
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.common.exceptions import (
    TimeoutException,
    NoSuchElementException,
    StaleElementReferenceException,
)
from constants import (
    LOAD_MORE_BUTTON_SELECTOR,
    ITEM_LINK_SELECTOR,
    DETAIL_SECTION_SELECTOR,
    DETAIL_ROW_SELECTOR,
    ROW_TITLE_SELECTOR,
    ROW_VALUE_SELECTOR,
    SCROLL_DELAY,
)
```

- time: Provides various time-related functions.
- selenium.webdriver.common.by.By: Specifies different locator strategies.
- selenium.webdriver.support.ui.WebDriverWait: Allows waiting for a certain condition to be true.
- selenium.webdriver.support.expected_conditions as EC: Defines expected conditions to use with WebDriverWait.
- selenium.common.exceptions: Provides various exceptions to handle different WebDriver errors.

- constants: Imports various constants such as CSS selectors and delay times.

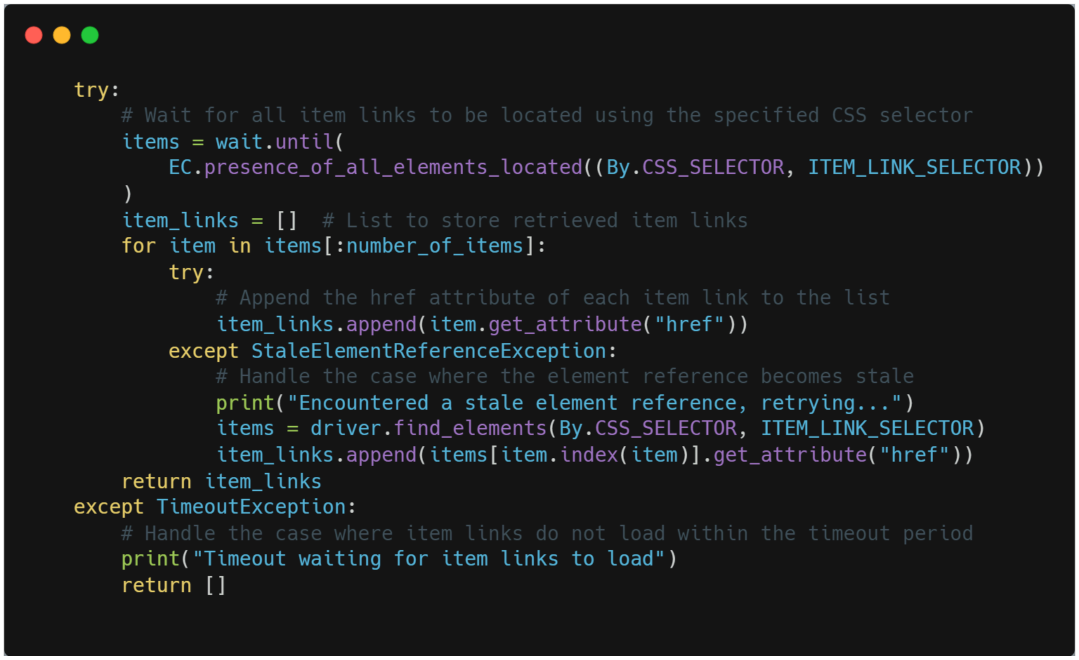
Function: retrieve_item_links(driver, number_of_items)

The retrieve_item_links function retrieves a specified number of item links from the main page.



```python
def retrieve_item_links(driver, number_of_items):
    """
    Retrieve a batch of item links from the main page.

    Parameters:
        driver (webdriver.Chrome): The WebDriver instance used for scraping.
        number_of_items (int): The number of item links to retrieve.

    Returns:
        list: A list of item links retrieved from the page.
    """
    wait = WebDriverWait(driver, 10)  # Wait up to 10 seconds for elements to be present
```

- Parameters:
  - driver: The WebDriver instance used for scraping.
  - number_of_items: The number of item links to retrieve.
- Returns: A list of item links retrieved from the page.

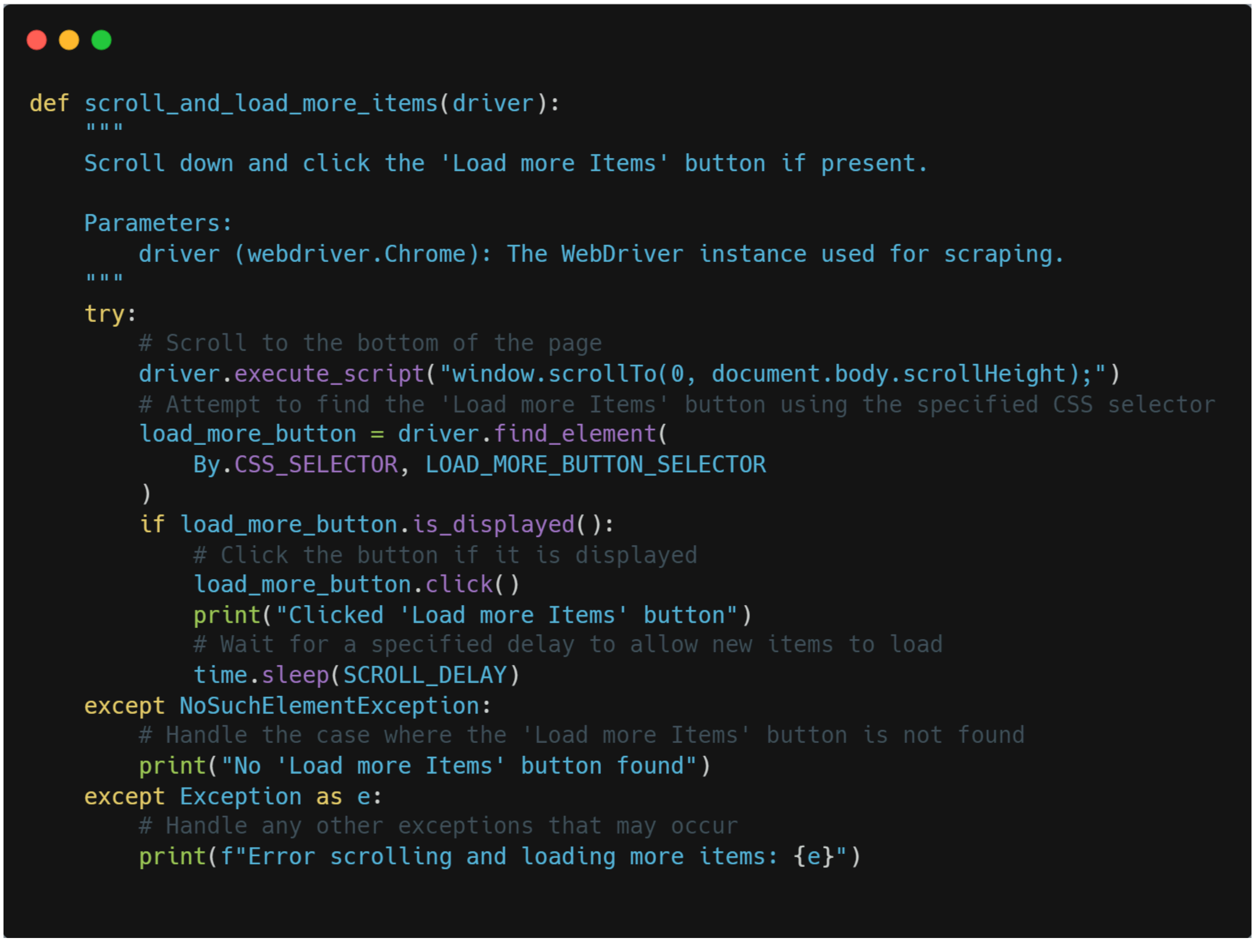


```python
    try:
        # Wait for all item links to be located using the specified CSS selector
        items = wait.until(
            EC.presence_of_all_elements_located((By.CSS_SELECTOR, ITEM_LINK_SELECTOR))
        )
        item_links = []  # List to store retrieved item links
        for item in items[:number_of_items]:
            try:
                # Append the href attribute of each item link to the list
                item_links.append(item.get_attribute("href"))
            except StaleElementReferenceException:
                # Handle the case where the element reference becomes stale
                print("Encountered a stale element reference, retrying...")
                items = driver.find_elements(By.CSS_SELECTOR, ITEM_LINK_SELECTOR)
                item_links.append(items[item.index(item)].get_attribute("href"))
        return item_links
    except TimeoutException:
        # Handle the case where item links do not load within the timeout period
        print("Timeout waiting for item links to load")
        return []
```

- WebDriverWait: Waits up to 10 seconds for the item links to be present.
- StaleElementReferenceException: Handles cases where the reference to an element becomes stale.

Function: scroll_and_load_more_items(driver)

The scroll_and_load_more_items function scrolls down the page and clicks the 'Load more Items' button if present.

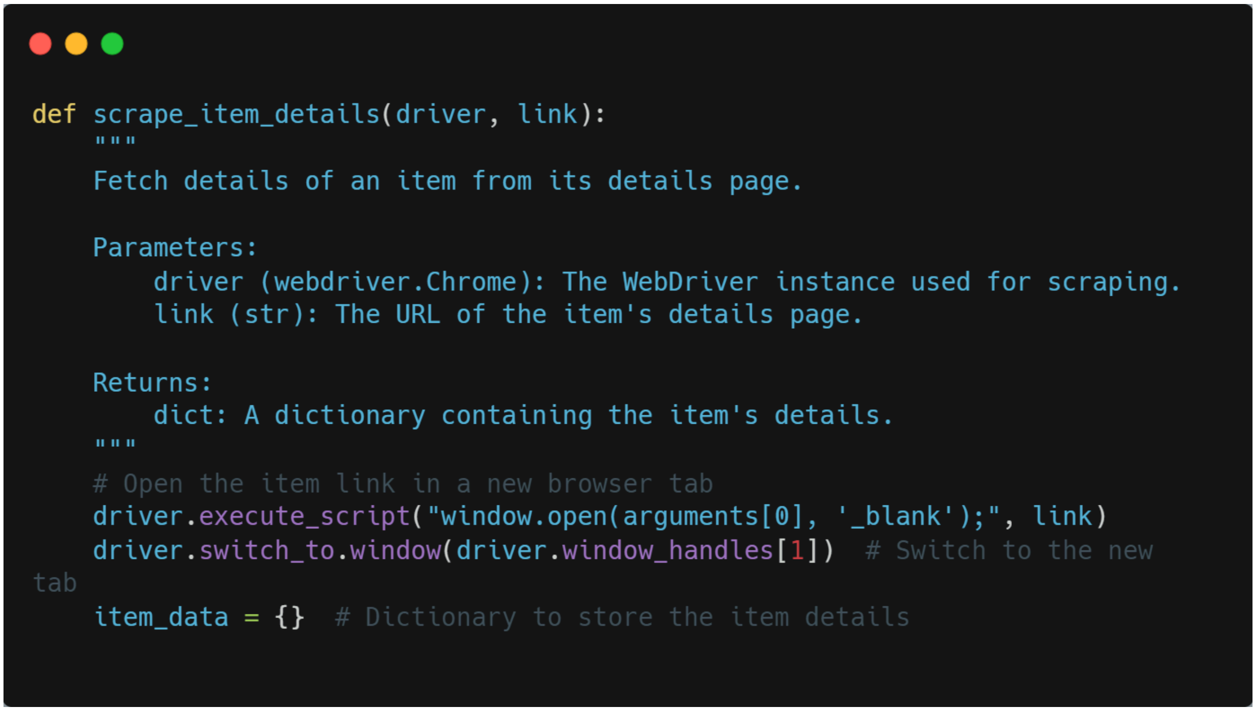


```python
def scroll_and_load_more_items(driver):
    """
    Scroll down and click the 'Load more Items' button if present.

    Parameters:
        driver (webdriver.Chrome): The WebDriver instance used for scraping.
    """
    try:
        # Scroll to the bottom of the page
        driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")
        # Attempt to find the 'Load more Items' button using the specified CSS selector
        load_more_button = driver.find_element(
            By.CSS_SELECTOR, LOAD_MORE_BUTTON_SELECTOR
        )
        if load_more_button.is_displayed():
            # Click the button if it is displayed
            load_more_button.click()
            print("Clicked 'Load more Items' button")
            # Wait for a specified delay to allow new items to load
            time.sleep(SCROLL_DELAY)
    except NoSuchElementException:
        # Handle the case where the 'Load more Items' button is not found
        print("No 'Load more Items' button found")
    except Exception as e:
        # Handle any other exceptions that may occur
        print(f"Error scrolling and loading more items: {e}")
```

- Parameters:
    - driver: The WebDriver instance used for scraping.
- Functionality:
    - Scrolls to the bottom of the page.
    - Attempts to find and click the 'Load more Items' button.
    - Waits for a specified delay to allow new items to load.
    - Handles exceptions if the button is not found or other errors occur.

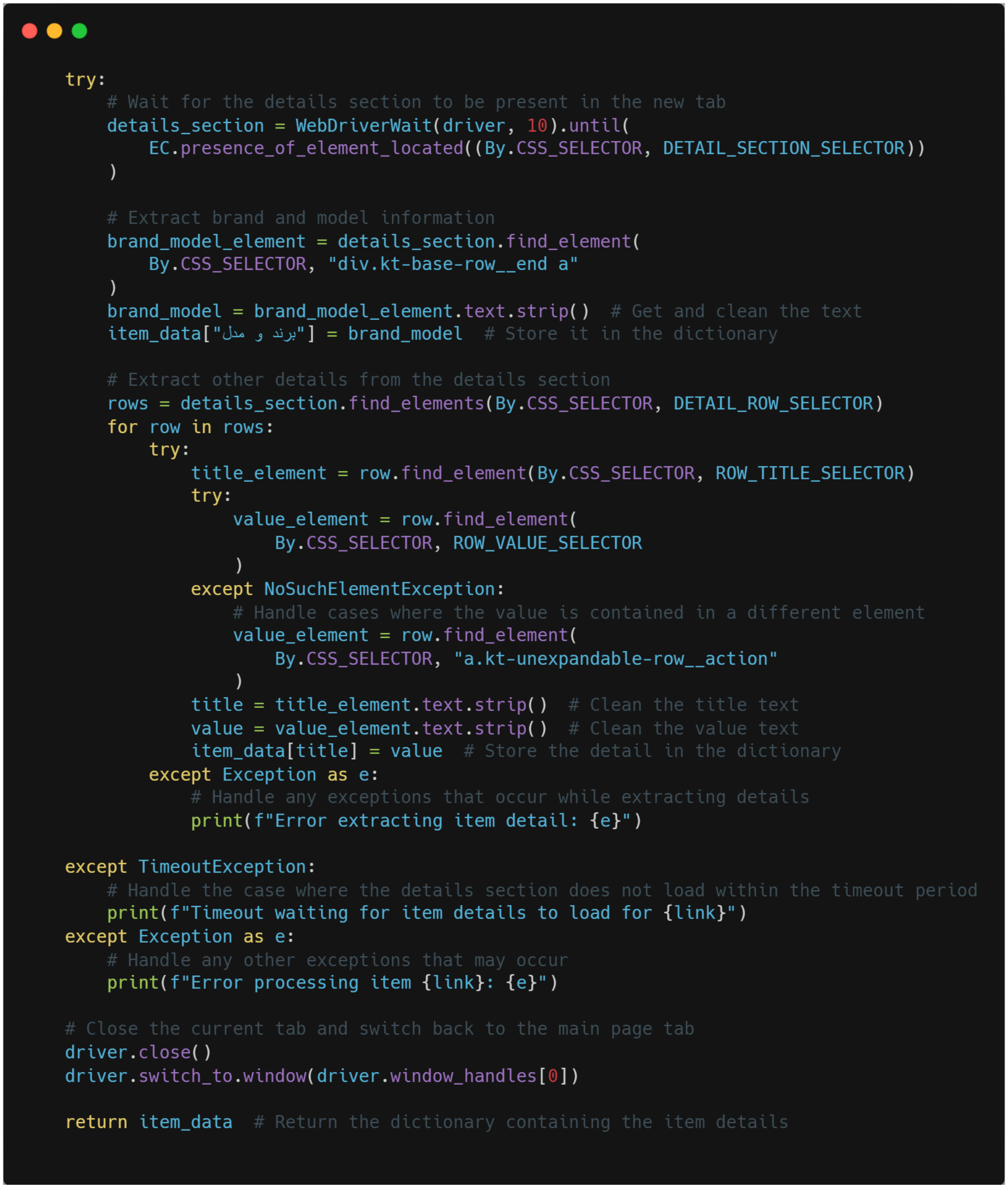
Function: scrape_item_details(driver, link)

The scrape_item_details function fetches the details of an item from its details page.



```
def scrape_item_details(driver, link):
    """
    Fetch details of an item from its details page.

    Parameters:
        driver (webdriver.Chrome): The WebDriver instance used for scraping.
        link (str): The URL of the item's details page.

    Returns:
        dict: A dictionary containing the item's details.
    """
    # Open the item link in a new browser tab
    driver.execute_script("window.open(arguments[0], '_blank');", link)
    driver.switch_to.window(driver.window_handles[1])  # Switch to the new
tab
    item_data = {}  # Dictionary to store the item details
```

- Parameters:
    - driver: The WebDriver instance used for scraping.
    - link: The URL of the item's details page.
- Returns: A dictionary containing the item's details.



```python
    try:
        # Wait for the details section to be present in the new tab
        details_section = WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.CSS_SELECTOR, DETAIL_SECTION_SELECTOR))
        )

        # Extract brand and model information
        brand_model_element = details_section.find_element(
            By.CSS_SELECTOR, "div.kt-base-row__end a"
        )
        brand_model = brand_model_element.text.strip()  # Get and clean the text
        item_data["برند و مدل"] = brand_model  # Store it in the dictionary

        # Extract other details from the details section
        rows = details_section.find_elements(By.CSS_SELECTOR, DETAIL_ROW_SELECTOR)
        for row in rows:
            try:
                title_element = row.find_element(By.CSS_SELECTOR, ROW_TITLE_SELECTOR)
                try:
                    value_element = row.find_element(
                        By.CSS_SELECTOR, ROW_VALUE_SELECTOR
                    )
                except NoSuchElementException:
                    # Handle cases where the value is contained in a different element
                    value_element = row.find_element(
                        By.CSS_SELECTOR, "a.kt-unexpandable-row__action"
                    )
                title = title_element.text.strip()  # Clean the title text
                value = value_element.text.strip()  # Clean the value text
                item_data[title] = value  # Store the detail in the dictionary
            except Exception as e:
                # Handle any exceptions that occur while extracting details
                print(f"Error extracting item detail: {e}")

    except TimeoutException:
        # Handle the case where the details section does not load within the timeout period
        print(f"Timeout waiting for item details to load for {link}")
    except Exception as e:
        # Handle any other exceptions that may occur
        print(f"Error processing item {link}: {e}")

    # Close the current tab and switch back to the main page tab
    driver.close()
    driver.switch_to.window(driver.window_handles[0])

    return item_data  # Return the dictionary containing the item details
```

- Functionality:
  - Opens the item link in a new browser tab.
  - Switches to the new tab.
  - Waits for the details section to be present.

    - Extracts brand, model, and other details.
    - Handles exceptions if elements are not found or other errors occur.
    - Closes the current tab and switches back to the main page tab.
    - Returns a dictionary containing the item details.

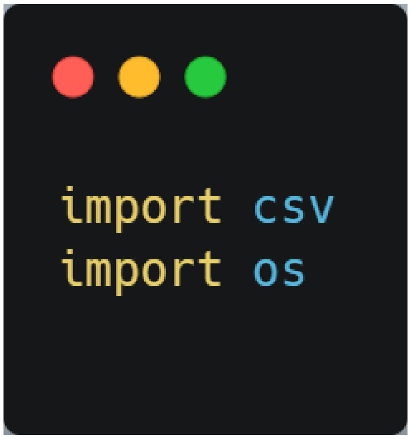
Explanation of Key Components

1. WebDriverWait:
    - Used to wait for elements to be present or visible before attempting to interact with them. This helps to handle dynamic content loading.
2. Exception Handling:
    - Different exceptions like TimeoutException, NoSuchElementException, and StaleElementReferenceException are handled to ensure the script can recover from common issues during scraping.
3. Dictionary for Item Details:
    - Details of each item are stored in a dictionary, making it easy to format and write to a CSV file later.

Module Documentation: csv_handler.py

This module handles the creation, writing, and management of CSV files used in the web scraping project. It includes a CSVWriter class and a factory function to initialize instances of this class.

Import Statements



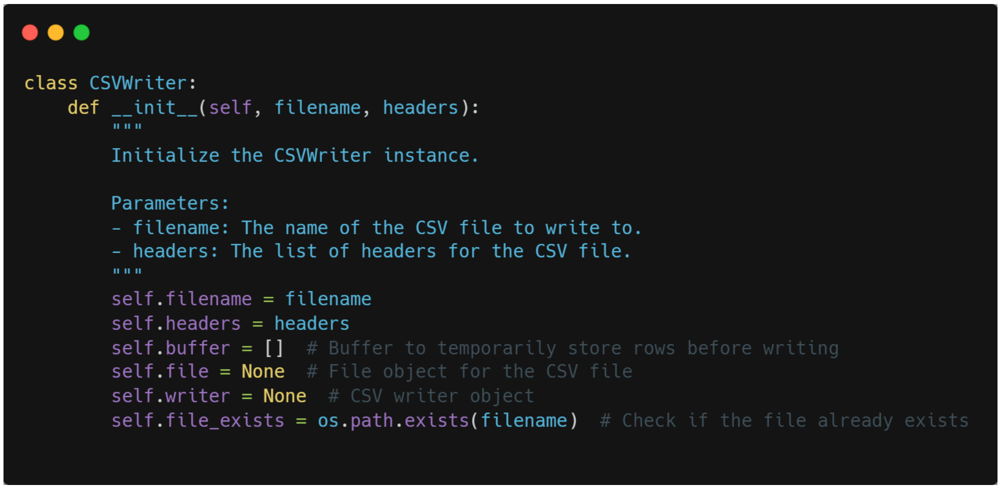
```
import csv
import os
```

- csv: Provides functionality to read from and write to CSV files.
- os: Used to interact with the operating system, particularly for file existence checks.

Class: CSVWriter

The CSVWriter class manages the writing of data to a CSV file, including buffering rows and ensuring the headers are written correctly.

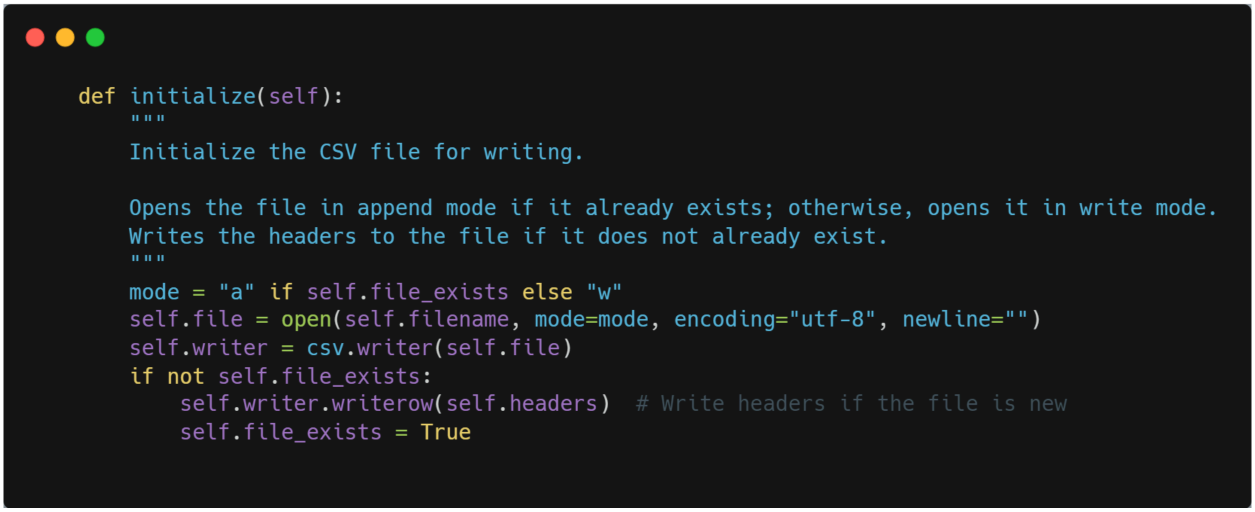
Initialization: __init__



```
class CSVWriter:
    def __init__(self, filename, headers):
        """
        Initialize the CSVWriter instance.

        Parameters:
        - filename: The name of the CSV file to write to.
        - headers: The list of headers for the CSV file.
        """
        self.filename = filename
        self.headers = headers
        self.buffer = []  # Buffer to temporarily store rows before writing
        self.file = None  # File object for the CSV file
        self.writer = None  # CSV writer object
        self.file_exists = os.path.exists(filename)  # Check if the file already exists
```

- Parameters:
    - filename: The name of the CSV file to write to.
    - headers: The list of headers for the CSV file.
- Attributes:
    - buffer: Temporarily stores rows before they are written to the file.
    - file: The file object for the CSV file.
    - writer: The CSV writer object.
    - file_exists: Checks if the file already exists to handle append vs write mode.

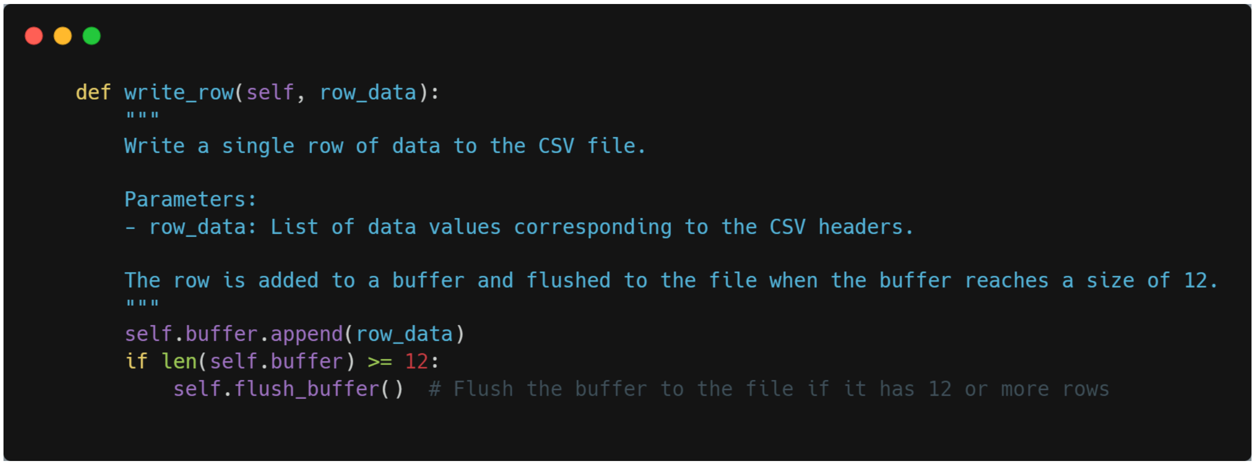
Method: initialize



```
def initialize(self):
    """
    Initialize the CSV file for writing.

    Opens the file in append mode if it already exists; otherwise, opens it in write mode.
    Writes the headers to the file if it does not already exist.
    """
    mode = "a" if self.file_exists else "w"
    self.file = open(self.filename, mode=mode, encoding="utf-8", newline="")
    self.writer = csv.writer(self.file)
    if not self.file_exists:
        self.writer.writerow(self.headers)  # Write headers if the file is new
        self.file_exists = True
```

- Functionality:
    - Opens the file in append mode ("a") if it exists, otherwise in write mode ("w").
    - Writes the headers to the file if it is new.
    - Sets up the CSV writer object.

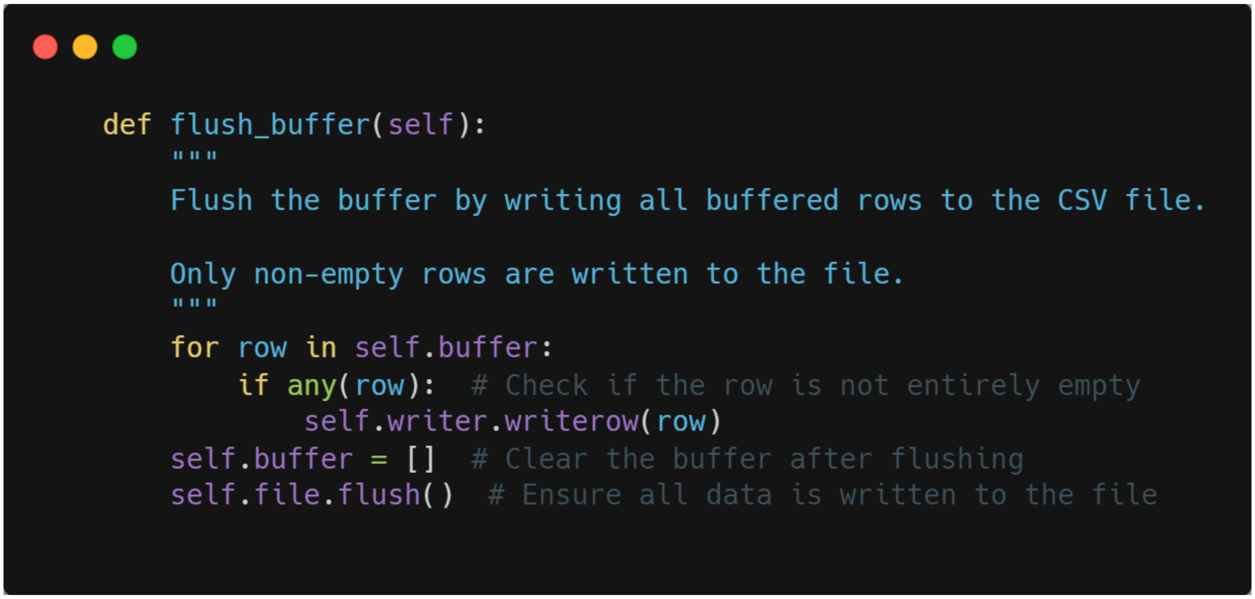
Method: write_row



```python
def write_row(self, row_data):
    """
    Write a single row of data to the CSV file.

    Parameters:
    - row_data: List of data values corresponding to the CSV headers.

    The row is added to a buffer and flushed to the file when the buffer reaches a size of 12.
    """
    self.buffer.append(row_data)
    if len(self.buffer) >= 12:
        self.flush_buffer()  # Flush the buffer to the file if it has 12 or more rows
```

- Parameters:
  - row_data: List of data values corresponding to the CSV headers.
- Functionality:
  - Adds the row to a buffer.
  - Flushes the buffer to the file when it reaches a size of 12 rows.

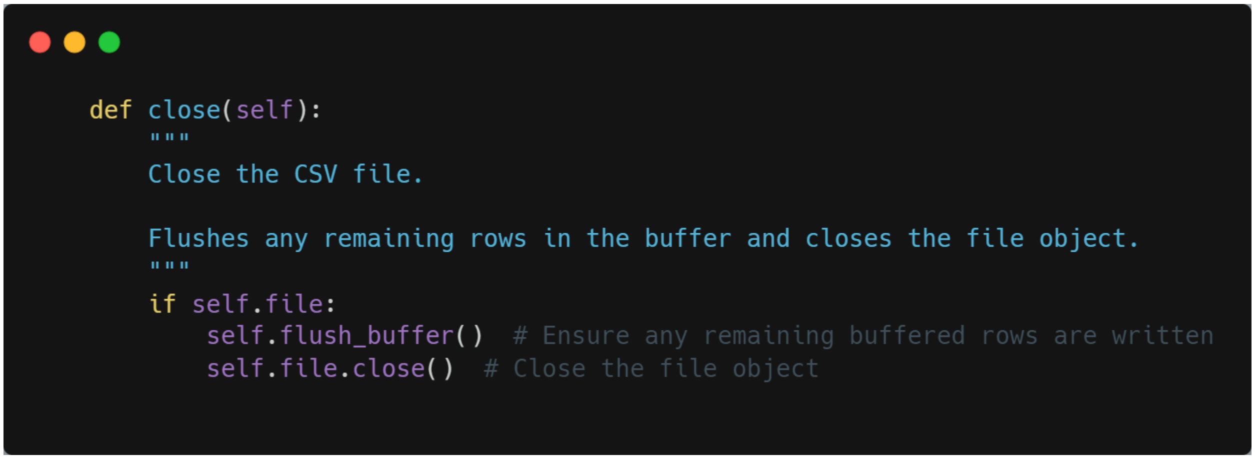
Method: flush_buffer



```python
def flush_buffer(self):
    """
    Flush the buffer by writing all buffered rows to the CSV file.

    Only non-empty rows are written to the file.
    """
    for row in self.buffer:
        if any(row):  # Check if the row is not entirely empty
            self.writer.writerow(row)
    self.buffer = []  # Clear the buffer after flushing
    self.file.flush()  # Ensure all data is written to the file
```

- Functionality:
  - Writes all buffered rows to the CSV file.
  - Only writes non-empty rows.
  - Clears the buffer after writing.
  - Flushes the file to ensure all data is written.

Method: close

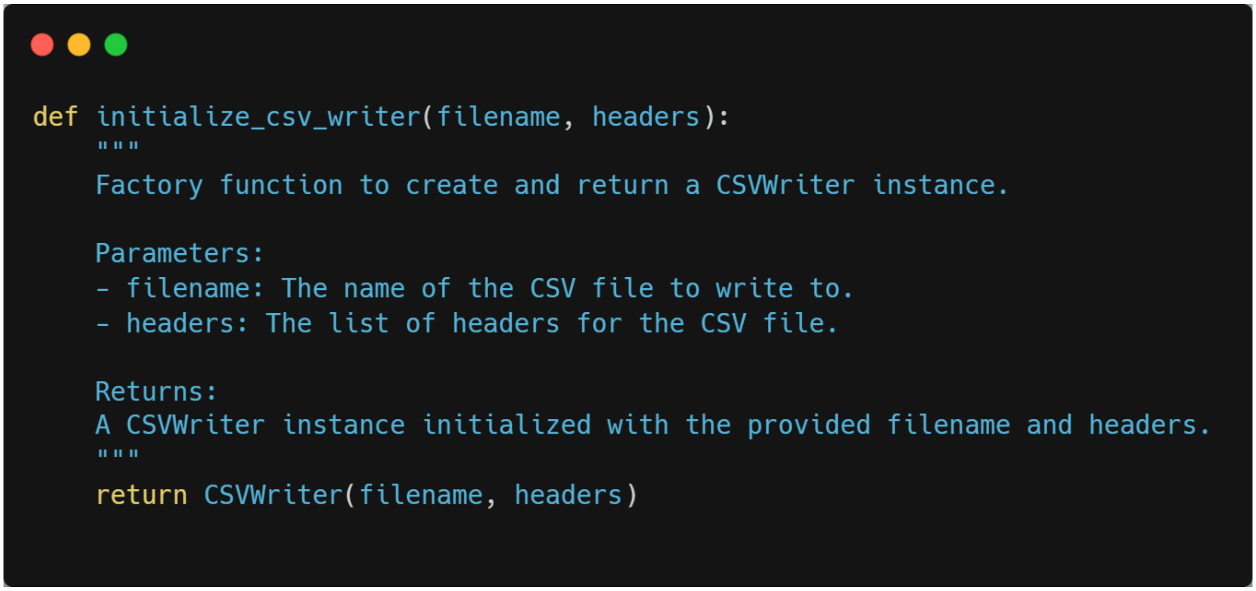


```
    def close(self):
        """
        Close the CSV file.

        Flushes any remaining rows in the buffer and closes the file object.
        """
        if self.file:
            self.flush_buffer()  # Ensure any remaining buffered rows are written
            self.file.close()  # Close the file object
```

- Functionality:
    - Flushes any remaining rows in the buffer.
    - Closes the file object.

Factory Function: initialize_csv_writer

The initialize_csv_writer function is a factory function to create and return a CSVWriter instance.



```
def initialize_csv_writer(filename, headers):
    """
    Factory function to create and return a CSVWriter instance.

    Parameters:
    - filename: The name of the CSV file to write to.
    - headers: The list of headers for the CSV file.

    Returns:
    A CSVWriter instance initialized with the provided filename and headers.
    """
    return CSVWriter(filename, headers)
```

- Parameters:
    - filename: The name of the CSV file to write to.
    - headers: The list of headers for the CSV file.

- Returns: A CSVWriter instance initialized with the provided filename and headers.

Key Components Explained

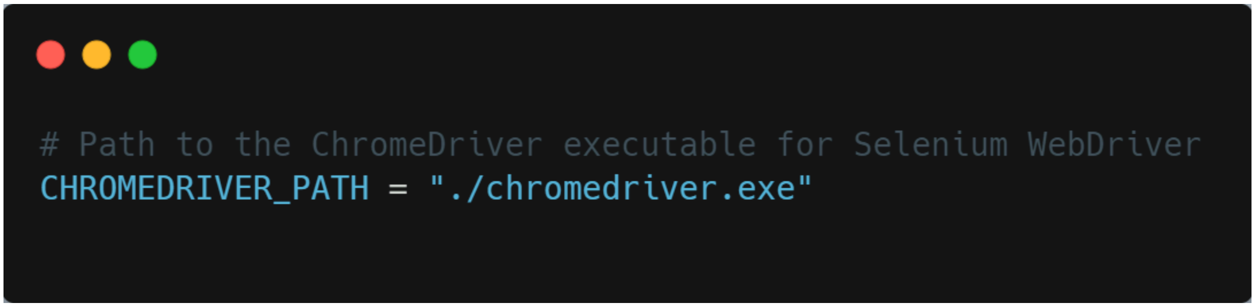
1. Buffering:
   - The CSVWriter uses a buffer to temporarily store rows before writing them to the file. This improves efficiency by reducing the number of write operations.
2. File Handling:
   - The class handles file opening in either append or write mode based on whether the file already exists.
   - Headers are written only if the file is new.
3. Flush Mechanism:
   - The buffer is flushed to the file when it reaches a size of 12 rows or when the close method is called.
   - Ensures all data is written to the file before closing.
4. Factory Function:
   - Provides a convenient way to create and initialize a CSVWriter instance with specified parameters.

Module Documentation: constants.py

This module contains the constants used throughout the web scraping project. These constants include configuration paths, URLs, CSV headers, scraping parameters, and CSS selectors necessary for interacting with the Divar website and processing data.

Constants

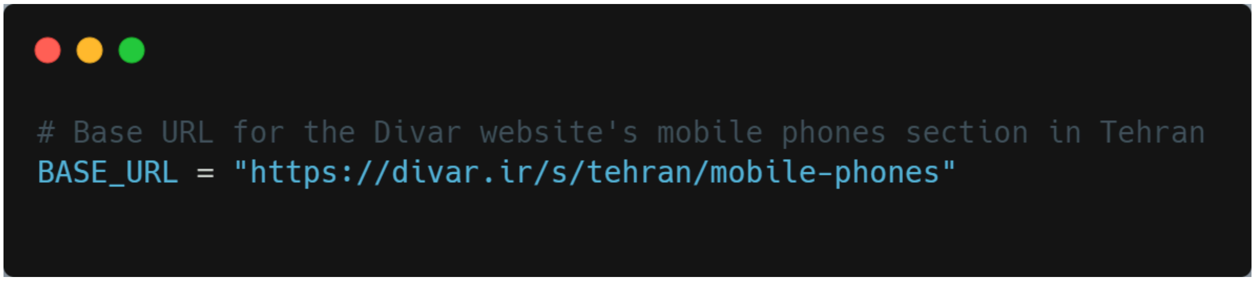
Path Constants



```
# Path to the ChromeDriver executable for Selenium WebDriver
CHROMEDRIVER_PATH = "./chromedriver.exe"
```

- CHROMEDRIVER_PATH: Specifies the path to the ChromeDriver executable, which is required for Selenium WebDriver to interact with the Chrome browser.

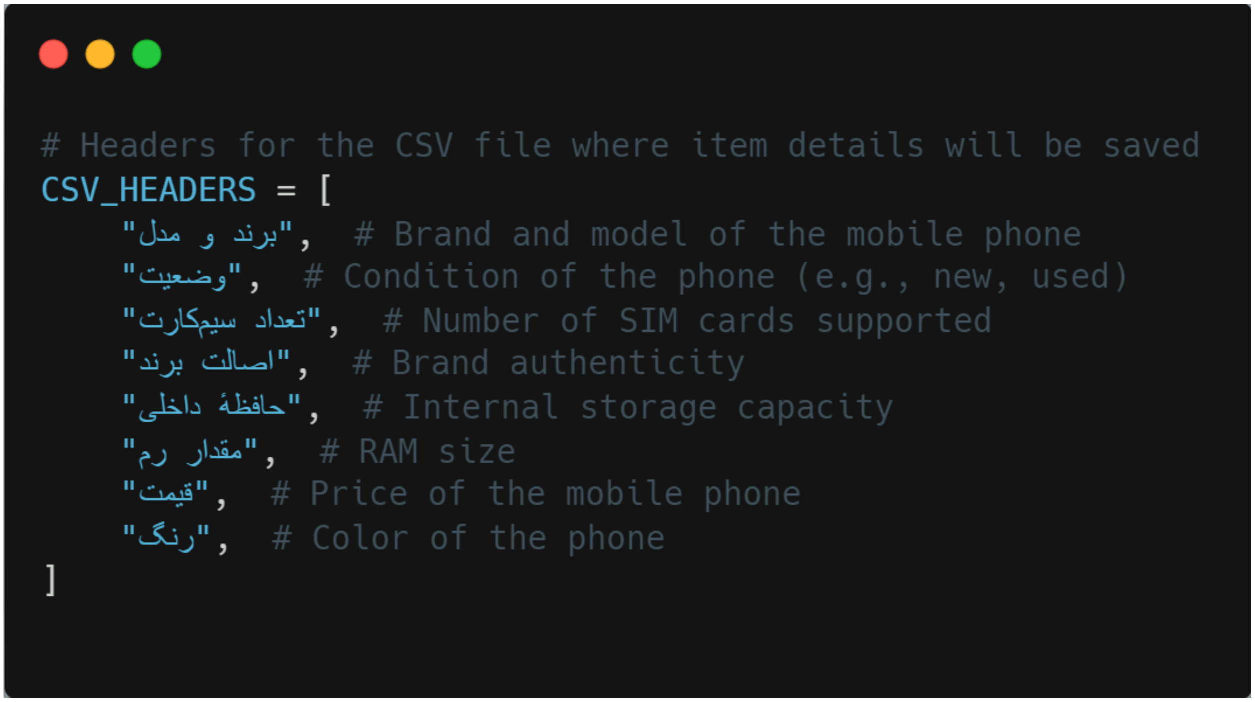
URL Constants



```
# Base URL for the Divar website's mobile phones section in Tehran
BASE_URL = "https://divar.ir/s/tehran/mobile-phones"
```

- BASE_URL: The base URL for the mobile phones section on the Divar website, specifically for Tehran. This is the starting point for scraping.

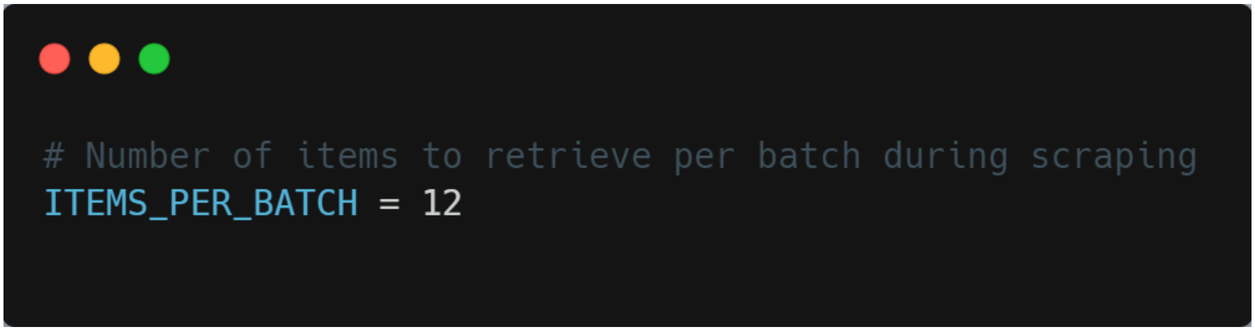
CSV File Constants



```
# Headers for the CSV file where item details will be saved
CSV_HEADERS = [
    "برند و مدل",  # Brand and model of the mobile phone
    "وضعیت",  # Condition of the phone (e.g., new, used)
    "تعداد سیمکارت",  # Number of SIM cards supported
    "اصالت برند",  # Brand authenticity
    "حافظهٔ داخلی",  # Internal storage capacity
    "مقدار رم",  # RAM size
    "قیمت",  # Price of the mobile phone
    "رنگ",  # Color of the phone
]
```

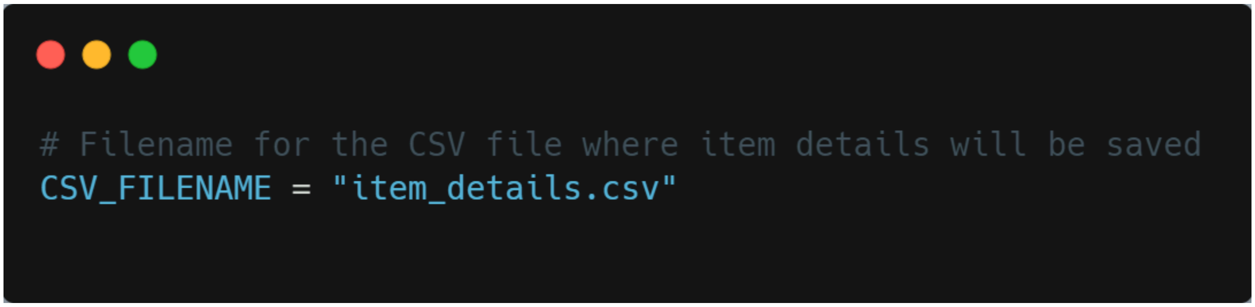
- CSV_HEADERS: A list of headers for the CSV file where the scraped item details will be saved. Each header corresponds to a specific attribute of the mobile phones being scraped:
    - برند و مدل: Brand and model of the mobile phone.
    - وضعیت: Condition of the phone (e.g., new, used).
    - تعداد سیمکارت: Number of SIM cards supported.
    - اصالت برند: Brand authenticity.
    - حافظهٔ داخلی: Internal storage capacity.
    - مقدار رم: RAM size.
    - قیمت: Price of the mobile phone.
    - رنگ: Color of the phone.

Scraping Parameters



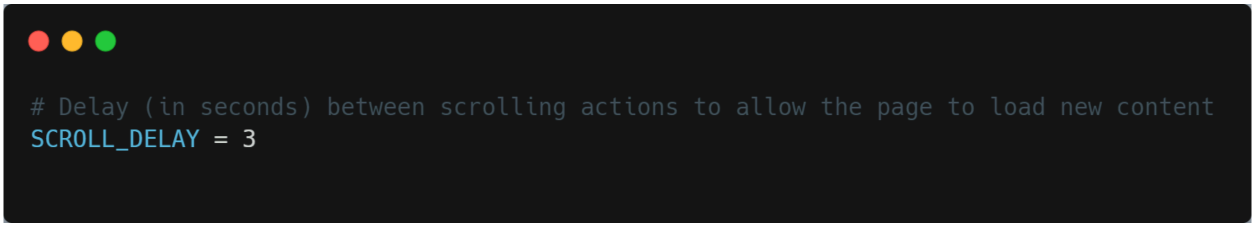
```
# Number of items to retrieve per batch during scraping
ITEMS_PER_BATCH = 12
```

- ITEMS_PER_BATCH: Defines the number of item links to retrieve per batch during the scraping process. This helps manage the workload and control the flow of data collection.



```
# Filename for the CSV file where item details will be saved
CSV_FILENAME = "item_details.csv"
```

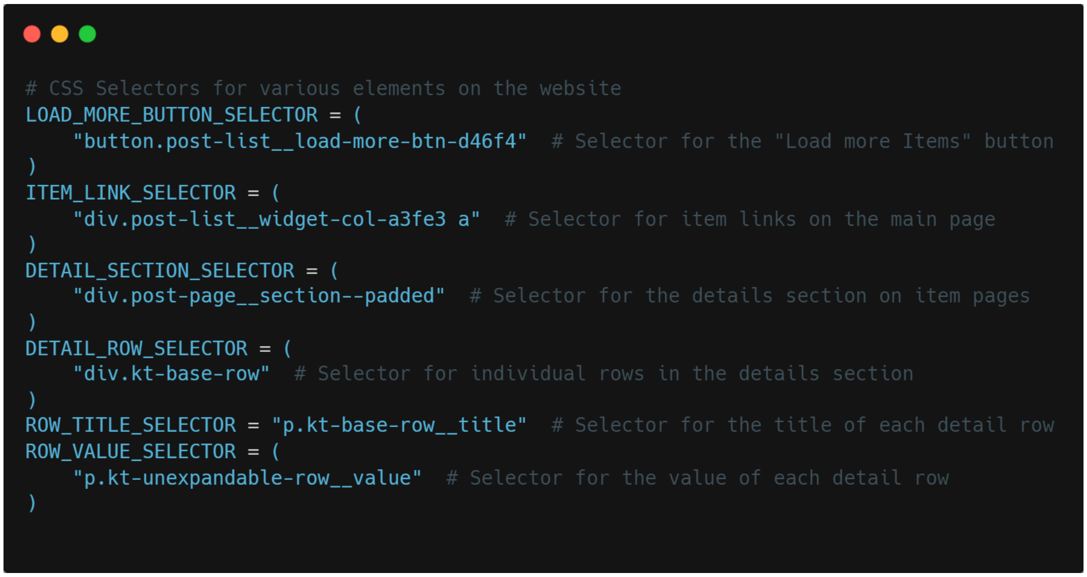
- CSV_FILENAME: The name of the CSV file where the scraped item details will be saved.



```
# Delay (in seconds) between scrolling actions to allow the page to load new content
SCROLL_DELAY = 3
```

- SCROLL_DELAY: Specifies the delay (in seconds) between scrolling actions to allow the page to load new content. This is important to avoid overloading the server and to ensure that new items have time to appear on the page.

CSS Selectors



```
# CSS Selectors for various elements on the website
LOAD_MORE_BUTTON_SELECTOR = (
    "button.post-list__load-more-btn-d46f4"  # Selector for the "Load more Items" button
)
ITEM_LINK_SELECTOR = (
    "div.post-list__widget-col-a3fe3 a"  # Selector for item links on the main page
)
DETAIL_SECTION_SELECTOR = (
    "div.post-page__section--padded"  # Selector for the details section on item pages
)
DETAIL_ROW_SELECTOR = (
    "div.kt-base-row"  # Selector for individual rows in the details section
)
ROW_TITLE_SELECTOR = "p.kt-base-row__title"  # Selector for the title of each detail row
ROW_VALUE_SELECTOR = (
    "p.kt-unexpandable-row__value"  # Selector for the value of each detail row
)
```

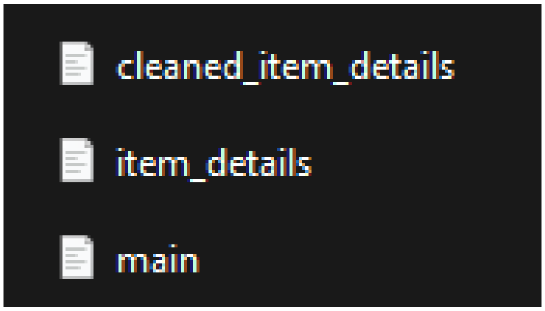
- LOAD_MORE_BUTTON_SELECTOR: CSS selector for the "Load more Items" button on the main page. This button is used to load additional items when the page is scrolled.
- ITEM_LINK_SELECTOR: CSS selector for item links on the main page. These links lead to individual item detail pages.
- DETAIL_SECTION_SELECTOR: CSS selector for the details section on item pages. This section contains all the detailed information about a specific item.
- DETAIL_ROW_SELECTOR: CSS selector for individual rows in the details section. Each row represents a different attribute of the item.
- ROW_TITLE_SELECTOR: CSS selector for the title of each detail row. The title indicates the type of information (e.g., "Price", "Brand").
- ROW_VALUE_SELECTOR: CSS selector for the value of each detail row. The value provides the specific details corresponding to the title.

Key Components Explained

1. File Paths and URLs:
   - CHROMEDRIVER_PATH and BASE_URL define the critical paths and starting point for the web scraping script.
2. CSV File Configuration:
   - CSV_HEADERS and CSV_FILENAME ensure that the data collected is stored in a structured and retrievable manner.
3. Scraping Parameters:
   - ITEMS_PER_BATCH and SCROLL_DELAY control the flow and performance of the scraping process, balancing efficiency with server load considerations.
4. CSS Selectors:
   - CSS selectors are essential for locating elements on the webpage. They enable the script to interact with the Divar website and extract the necessary information.

1.2 Data Cleaning and Transformation:

This folder Contains:



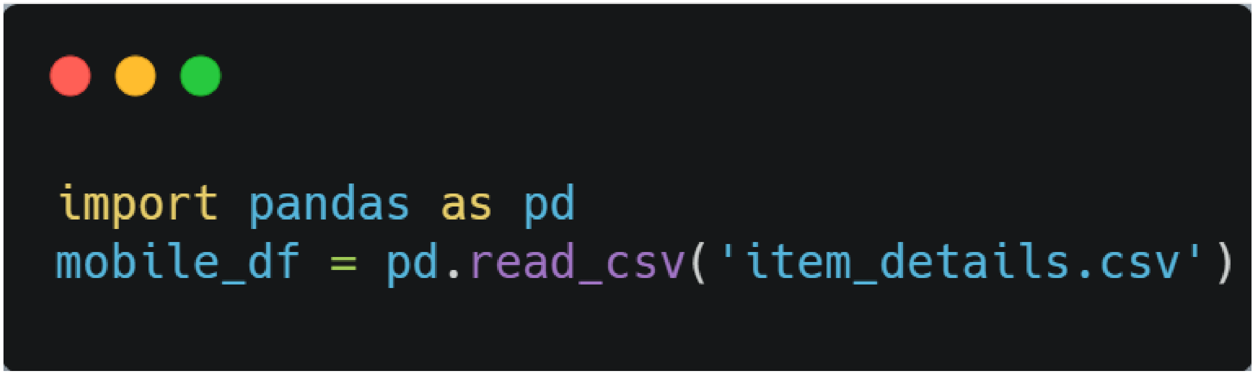
cleaned_item_details

item_details

main

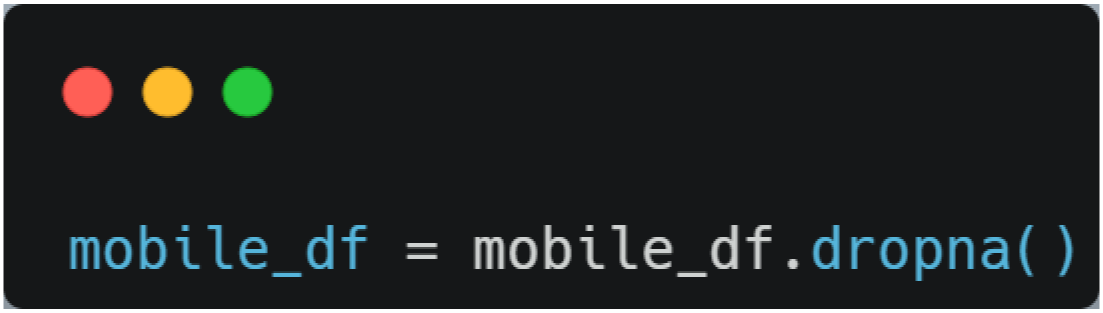
item_details.csv is our dataset mined from the last step.
cleaned_item_details.csv is our new dataset after cleaning.
main.ipynb is the file where we use various methods to clean our dataset.

main.ipynb:



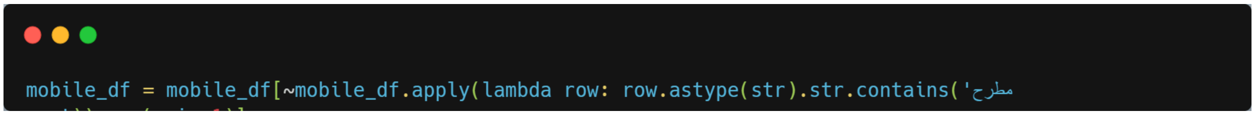
```
import pandas as pd
mobile_df = pd.read_csv('item_details.csv')
```

Use the pandas library to read item_details.csv

- 

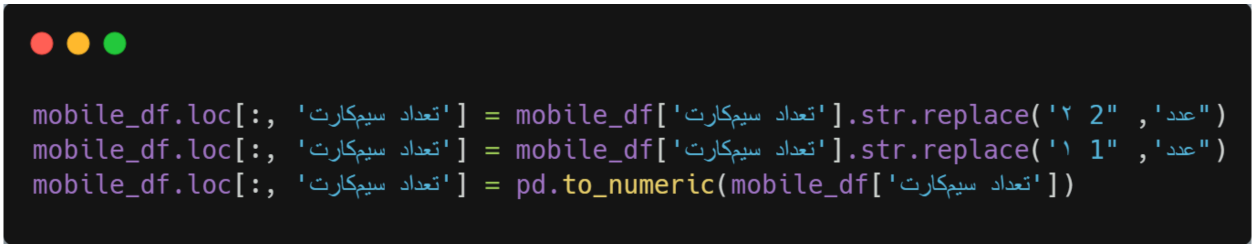
```
mobile_df = mobile_df.dropna()
```

Delete rows that contain NaN values.

- 

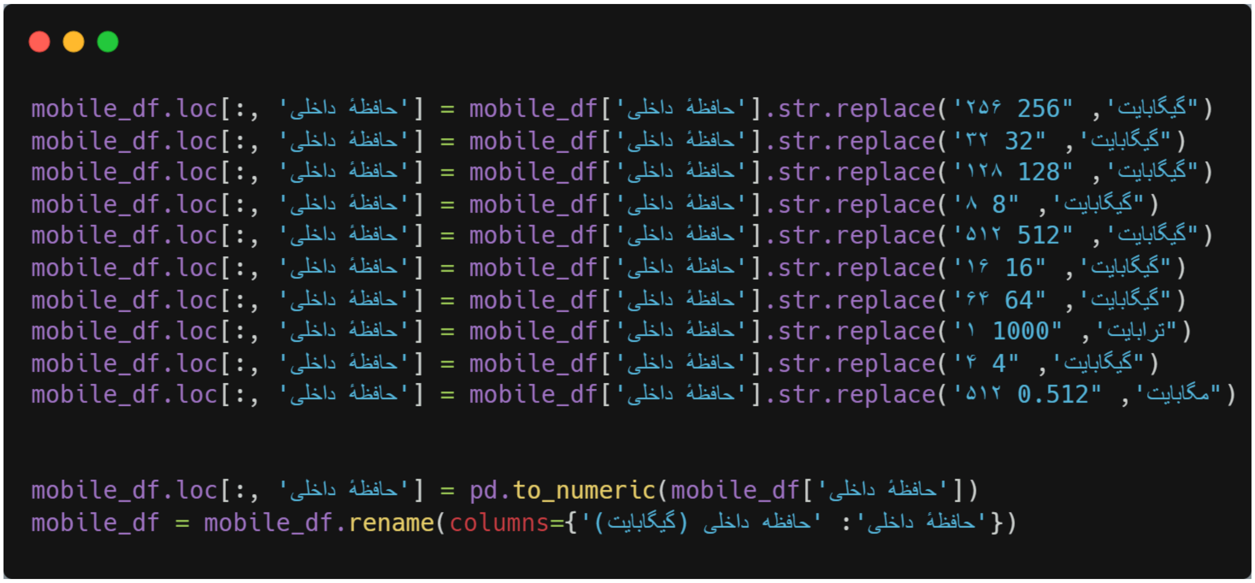
```
mobile_df = mobile_df[~mobile_df.apply(lambda row: row.astype(str).str.contains('مطرح
```

Delete rows that contain 'مطرح نیست' values.



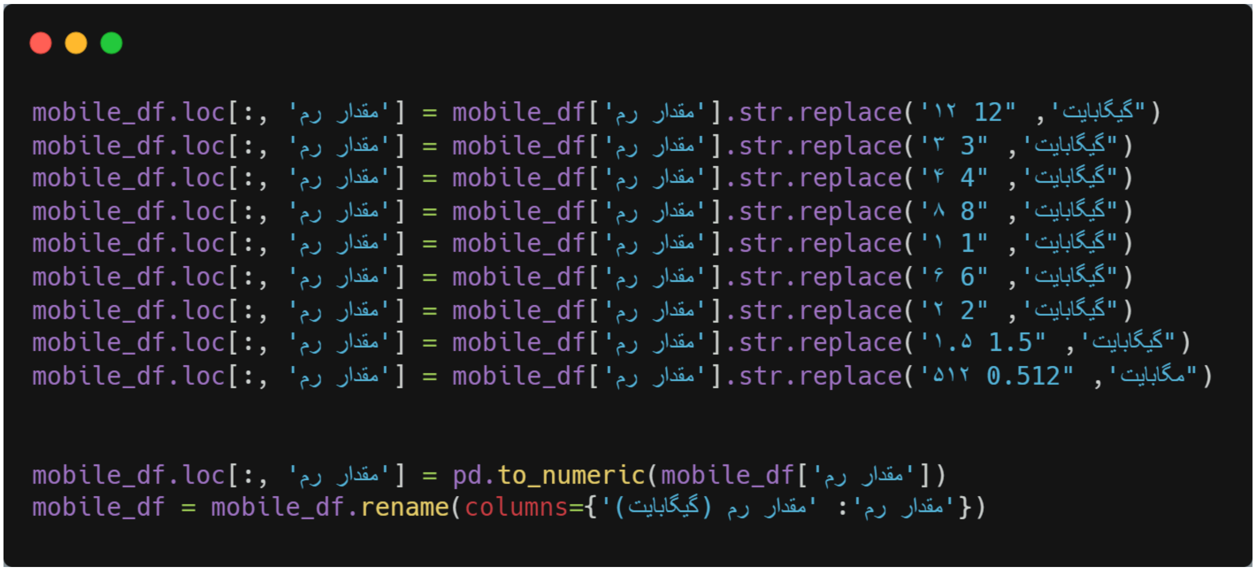
```
mobile_df.loc[:, 'تعداد سیمکارت'] = mobile_df['تعداد سیمکارت'].str.replace('۲ 2" ,'عدد")
mobile_df.loc[:, 'تعداد سیمکارت'] = mobile_df['تعداد سیمکارت'].str.replace('۱ 1" ,'عدد")
mobile_df.loc[:, 'تعداد سیمکارت'] = pd.to_numeric(mobile_df['تعداد سیمکارت'])
```

In the 'تعداد سیم کارت' column, remove the word 'عدد', convert Persian numbers to English numbers, and then convert them to numerical form.



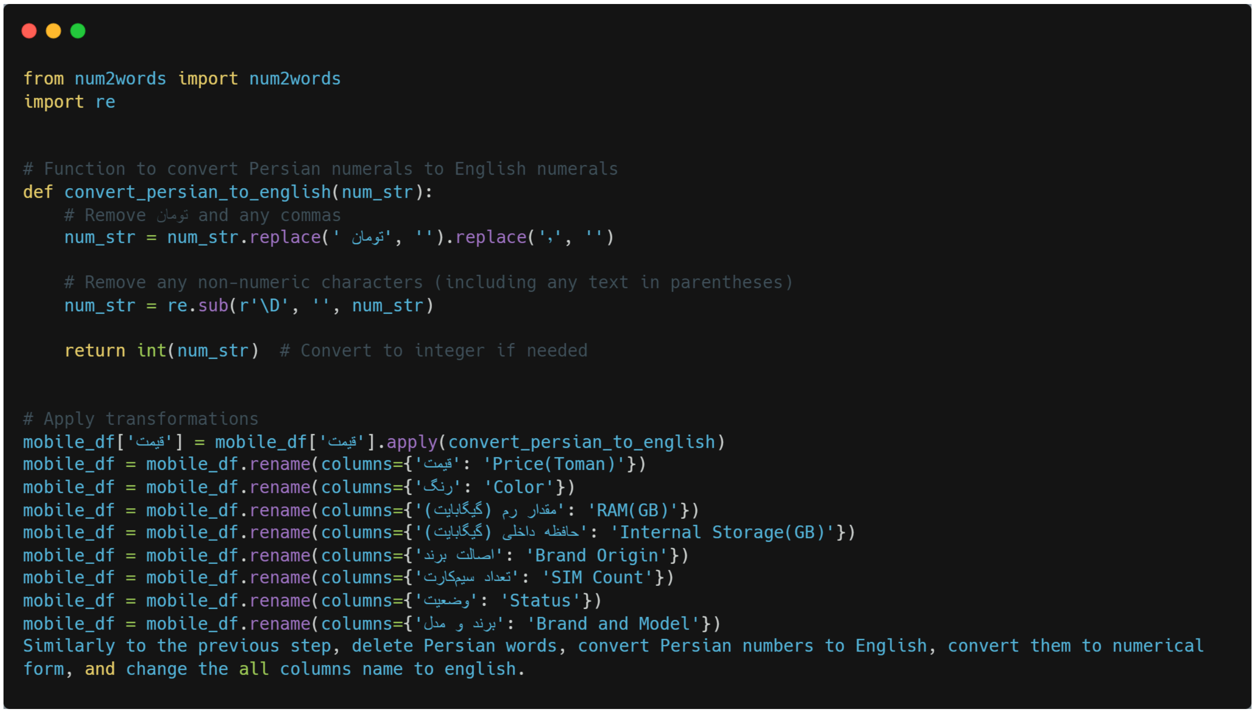
```
mobile_df.loc[:, 'حافظه داخلی'] = mobile_df['حافظه داخلی'].str.replace('۲۵۶ 256" ,'گیگابایت")
mobile_df.loc[:, 'حافظه داخلی'] = mobile_df['حافظه داخلی'].str.replace('۳۲ 32" ,'گیگابایت")
mobile_df.loc[:, 'حافظه داخلی'] = mobile_df['حافظه داخلی'].str.replace('۱۲۸ 128" ,'گیگابایت")
mobile_df.loc[:, 'حافظه داخلی'] = mobile_df['حافظه داخلی'].str.replace('۸ 8" ,'گیگابایت")
mobile_df.loc[:, 'حافظه داخلی'] = mobile_df['حافظه داخلی'].str.replace('۵۱۲ 512" ,'گیگابایت")
mobile_df.loc[:, 'حافظه داخلی'] = mobile_df['حافظه داخلی'].str.replace('۱۶ 16" ,'گیگابایت")
mobile_df.loc[:, 'حافظه داخلی'] = mobile_df['حافظه داخلی'].str.replace('۶۴ 64" ,'گیگابایت")
mobile_df.loc[:, 'حافظه داخلی'] = mobile_df['حافظه داخلی'].str.replace('۱ 1000" ,'ترابایت")
mobile_df.loc[:, 'حافظه داخلی'] = mobile_df['حافظه داخلی'].str.replace('۴ 4" ,'گیگابایت")
mobile_df.loc[:, 'حافظه داخلی'] = mobile_df['حافظه داخلی'].str.replace('۵۱۲ 0.512" ,'مگابایت")


mobile_df.loc[:, 'حافظه داخلی'] = pd.to_numeric(mobile_df['حافظه داخلی'])
mobile_df = mobile_df.rename(columns={'حافظه داخلی (گیگابایت)' :'حافظه داخلی'})
```

Similarly to the previous step, delete Persian words, convert Persian numbers to English, convert them to numerical form, and change the column name.

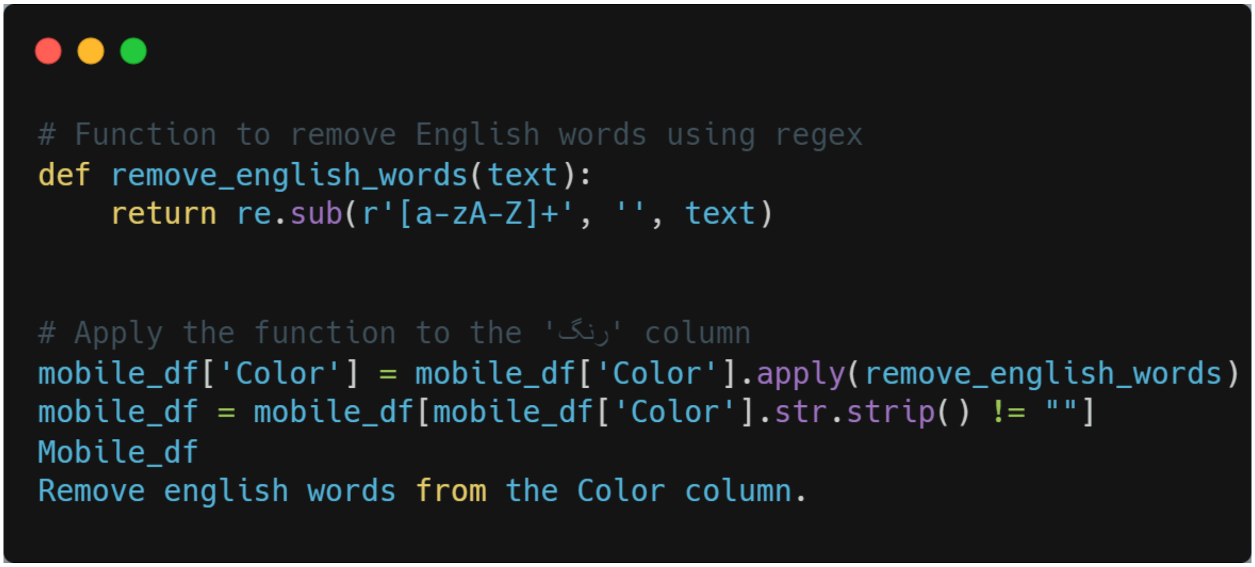


```
mobile_df.loc[:, 'مقدار رم'] = mobile_df['مقدار رم'].str.replace('۱۲ 12" ,'گیگابایت")
mobile_df.loc[:, 'مقدار رم'] = mobile_df['مقدار رم'].str.replace('۳ 3" ,'گیگابایت")
mobile_df.loc[:, 'مقدار رم'] = mobile_df['مقدار رم'].str.replace('۴ 4" ,'گیگابایت")
mobile_df.loc[:, 'مقدار رم'] = mobile_df['مقدار رم'].str.replace('۸ 8" ,'گیگابایت")
mobile_df.loc[:, 'مقدار رم'] = mobile_df['مقدار رم'].str.replace('۱ 1" ,'گیگابایت")
mobile_df.loc[:, 'مقدار رم'] = mobile_df['مقدار رم'].str.replace('۶ 6" ,'گیگابایت")
mobile_df.loc[:, 'مقدار رم'] = mobile_df['مقدار رم'].str.replace('۲ 2" ,'گیگابایت")
mobile_df.loc[:, 'مقدار رم'] = mobile_df['مقدار رم'].str.replace('۱.۵ 1.5" ,'گیگابایت")
mobile_df.loc[:, 'مقدار رم'] = mobile_df['مقدار رم'].str.replace('۵۱۲ 0.512" ,'مگابایت")


mobile_df.loc[:, 'مقدار رم'] = pd.to_numeric(mobile_df['مقدار رم'])
mobile_df = mobile_df.rename(columns={'مقدار رم (گیگابایت)' :'مقدار رم'})
```

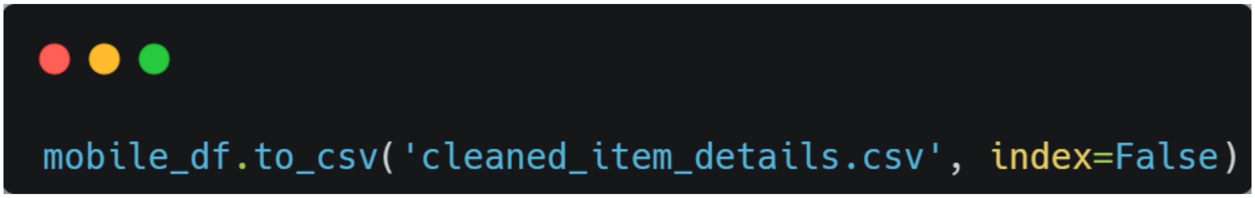
Similarly to the previous step, delete Persian words, convert Persian numbers to English, convert them to numerical form, and change the column name.



```
from num2words import num2words
import re


# Function to convert Persian numerals to English numerals
def convert_persian_to_english(num_str):
    # Remove تومان and any commas
    num_str = num_str.replace(' تومان', '').replace(',', '')

    # Remove any non-numeric characters (including any text in parentheses)
    num_str = re.sub(r'\D', '', num_str)

    return int(num_str)  # Convert to integer if needed


# Apply transformations
mobile_df['قیمت'] = mobile_df['قیمت'].apply(convert_persian_to_english)
mobile_df = mobile_df.rename(columns={'قیمت': 'Price(Toman)'})
mobile_df = mobile_df.rename(columns={'رنگ': 'Color'})
mobile_df = mobile_df.rename(columns={'مقدار رم (گیگابایت)': 'RAM(GB)'})
mobile_df = mobile_df.rename(columns={'حافظه داخلی (گیگابایت)': 'Internal Storage(GB)'})
mobile_df = mobile_df.rename(columns={'اصالت برند': 'Brand Origin'})
mobile_df = mobile_df.rename(columns={'تعداد سیمکارت': 'SIM Count'})
mobile_df = mobile_df.rename(columns={'وضعیت': 'Status'})
mobile_df = mobile_df.rename(columns={'برند و مدل': 'Brand and Model'})
Similarly to the previous step, delete Persian words, convert Persian numbers to English, convert them to numerical
form, and change the all columns name to english.
```

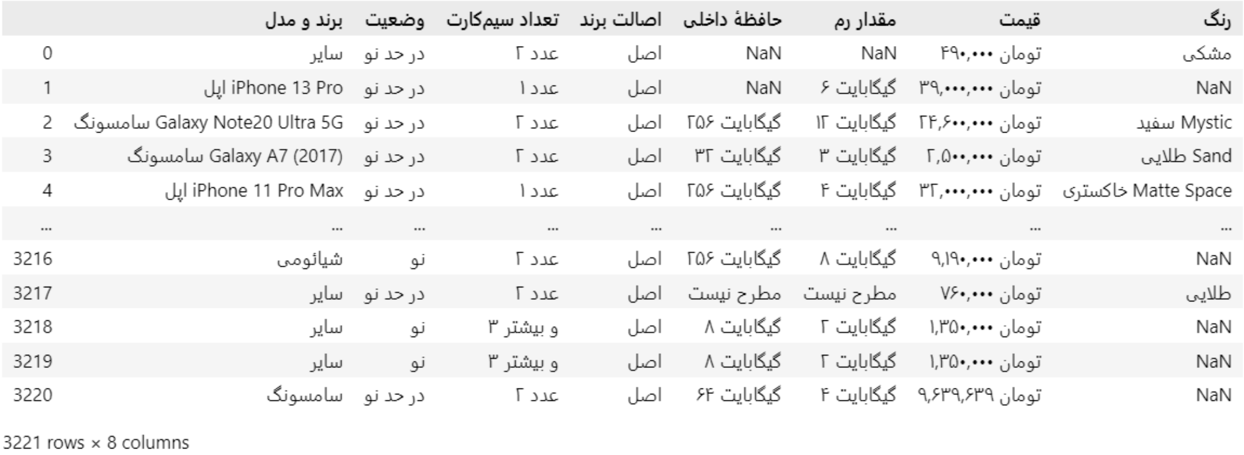


```
# Function to remove English words using regex
def remove_english_words(text):
    return re.sub(r'[a-zA-Z]+', '', text)


# Apply the function to the 'رنگ' column
mobile_df['Color'] = mobile_df['Color'].apply(remove_english_words)
mobile_df = mobile_df[mobile_df['Color'].str.strip() != ""]
Mobile_df
Remove english words from the Color column.
```

- 

```
mobile_df.to_csv('cleaned_item_details.csv', index=False)
```

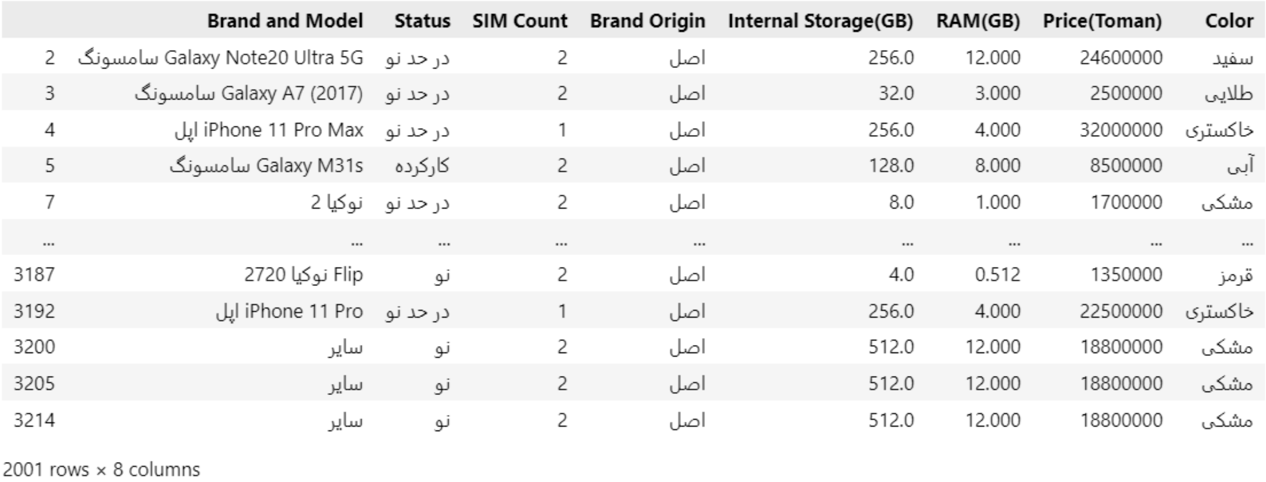
Finally, we cleaned our data and stored it in the cleaned_item_details.csv file.



	برند و مدل	وضعیت	تعداد سیم‌کارت	اصالت برند	حافظهٔ داخلی	مقدار رم	قیمت	رنگ
0	سایر	در حد نو	۲ عدد	اصل	NaN	NaN	۴۹۰,۰۰۰ تومان	مشکی
1	iPhone 13 Pro اپل	در حد نو	۱ عدد	اصل	NaN	۶ گیگابایت	۳۹,۰۰۰,۰۰۰ تومان	NaN
2	Galaxy Note20 Ultra 5G سامسونگ	در حد نو	۲ عدد	اصل	۲۵۶ گیگابایت	۱۲ گیگابایت	۲۴,۶۰۰,۰۰۰ تومان	Mystic سفید
3	Galaxy A7 (2017) سامسونگ	در حد نو	۲ عدد	اصل	۳۲ گیگابایت	۳ گیگابایت	۲,۵۰۰,۰۰۰ تومان	Sand طلایی
4	iPhone 11 Pro Max اپل	در حد نو	۱ عدد	اصل	۲۵۶ گیگابایت	۴ گیگابایت	۳۲,۰۰۰,۰۰۰ تومان	Matte Space خاکستری
...	...	...	...	...	...	...	...	...
3216	شیائومی	نو	۲ عدد	اصل	۲۵۶ گیگابایت	۸ گیگابایت	۹,۱۹۰,۰۰۰ تومان	NaN
3217	سایر	در حد نو	۲ عدد	اصل	مطرح نیست	مطرح نیست	۷۶۰,۰۰۰ تومان	طلایی
3218	سایر	نو	۳ و بیشتر	اصل	۸ گیگابایت	۲ گیگابایت	۱,۳۵۰,۰۰۰ تومان	NaN
3219	سایر	نو	۳ و بیشتر	اصل	۸ گیگابایت	۲ گیگابایت	۱,۳۵۰,۰۰۰ تومان	NaN
3220	سامسونگ	در حد نو	۲ عدد	اصل	۶۴ گیگابایت	۴ گیگابایت	۹,۶۳۹,۶۳۹ تومان	NaN

3221 rows × 8 columns

item_details.csv



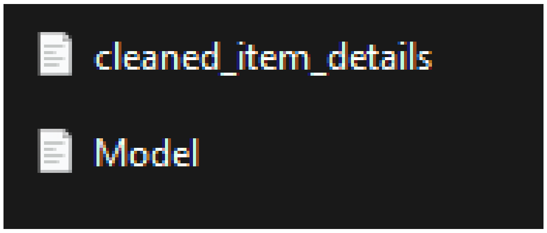
	Brand and Model	Status	SIM Count	Brand Origin	Internal Storage(GB)	RAM(GB)	Price(Toman)	Color
2	Galaxy Note20 Ultra 5G سامسونگ	در حد نو	2	اصل	256.0	12.000	24600000	سفید
3	Galaxy A7 (2017) سامسونگ	در حد نو	2	اصل	32.0	3.000	2500000	طلایی
4	iPhone 11 Pro Max اپل	در حد نو	1	اصل	256.0	4.000	32000000	خاکستری
5	Galaxy M31s سامسونگ	کارکرده	2	اصل	128.0	8.000	8500000	آبی
7	نوکیا 2	در حد نو	2	اصل	8.0	1.000	1700000	مشکی
...	...	...	...	...	...	...	...	...
3187	نوکیا 2720 Flip	نو	2	اصل	4.0	0.512	1350000	قرمز
3192	iPhone 11 Pro اپل	در حد نو	1	اصل	256.0	4.000	22500000	خاکستری
3200	سایر	نو	2	اصل	512.0	12.000	18800000	مشکی
3205	سایر	نو	2	اصل	512.0	12.000	18800000	مشکی
3214	سایر	نو	2	اصل	512.0	12.000	18800000	مشکی

2001 rows × 8 columns

cleaned_item_details.csv

2.1 Linear Regression Model:

This folder Contains:

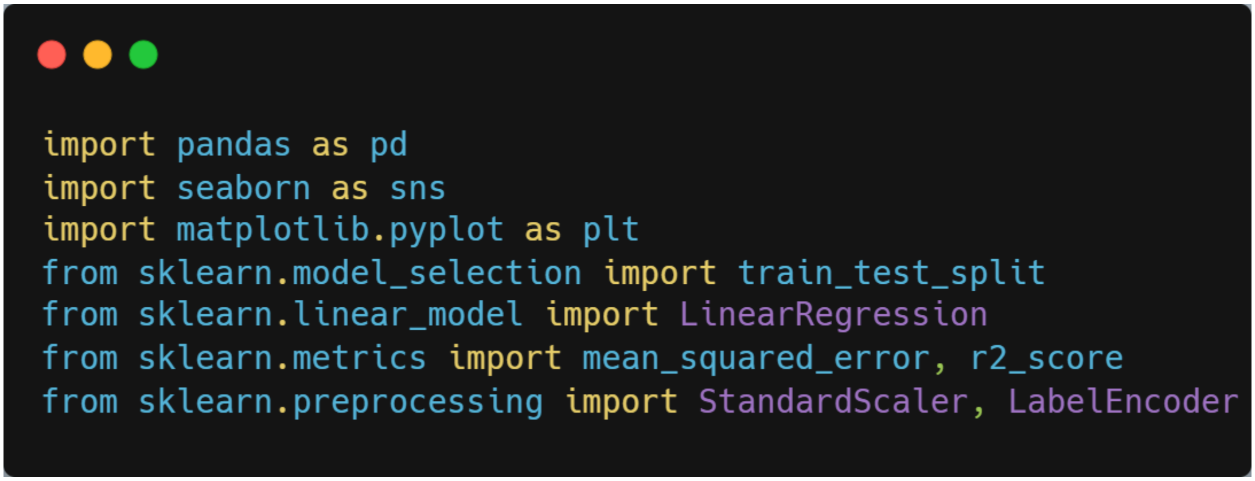


cleaned_item_details

Model

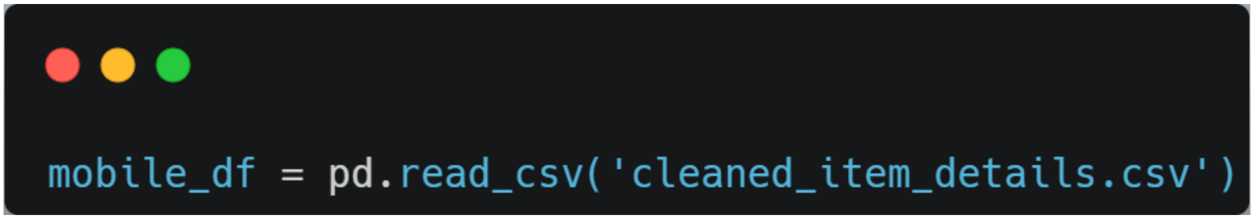
cleaned_item_details.csv is our new dataset after cleaning.
model.ipynb is the file where we implement our linear regression model.

model.ipynb:



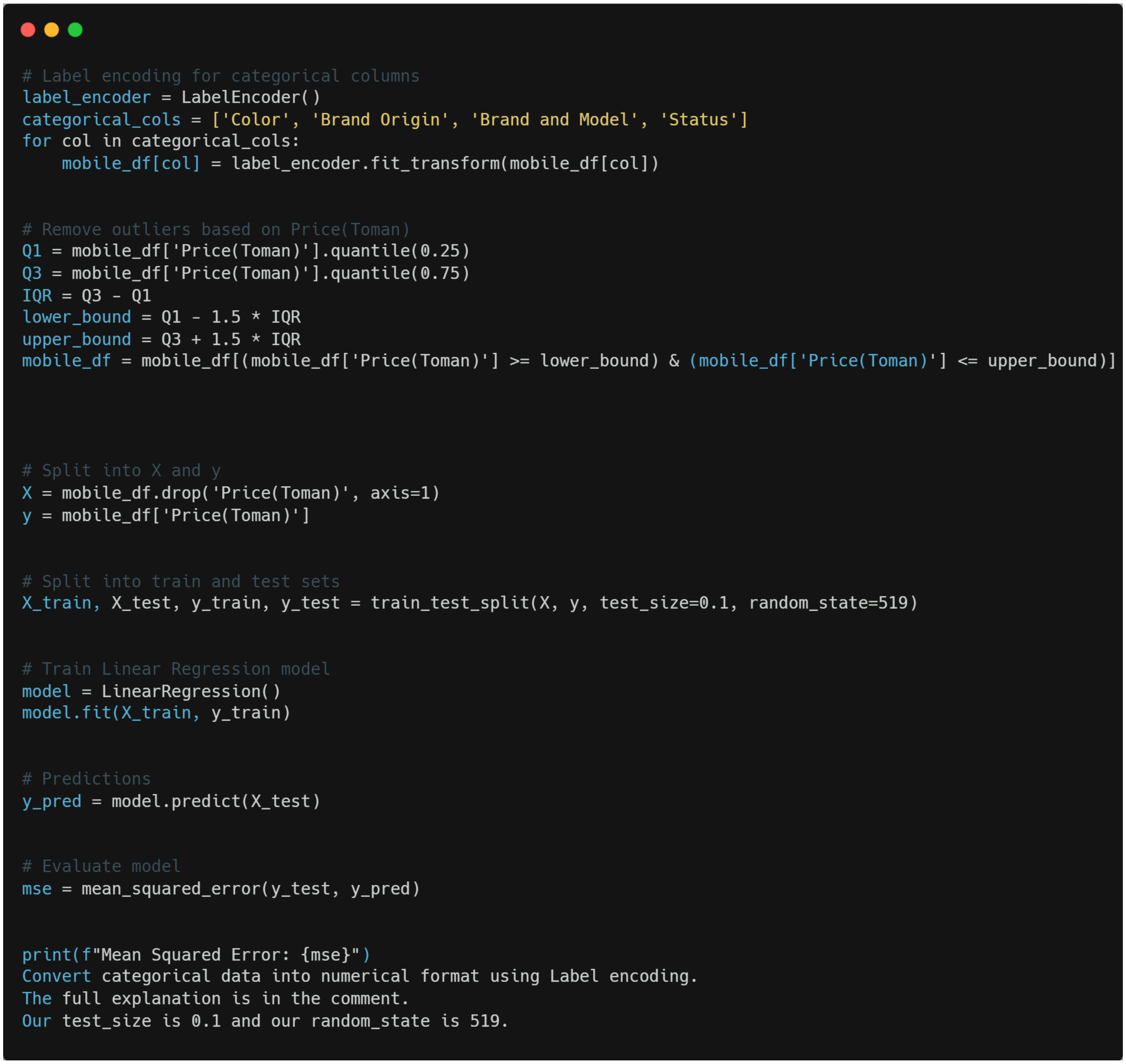
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import StandardScaler, LabelEncoder
```

Add libraries.

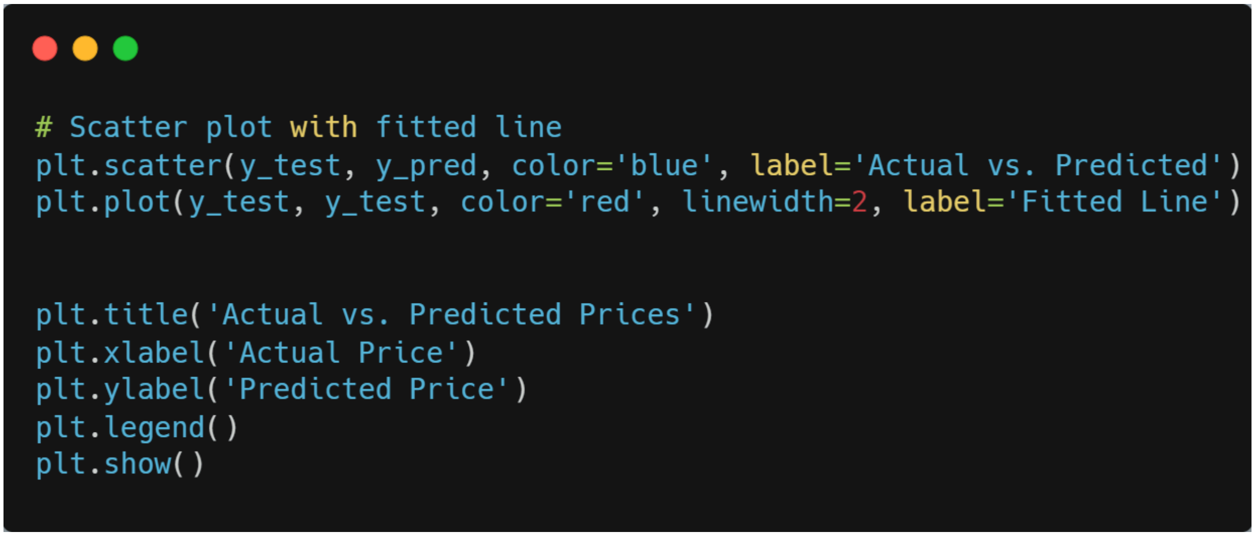


```
mobile_df = pd.read_csv('cleaned_item_details.csv')
```

read item_details.csv

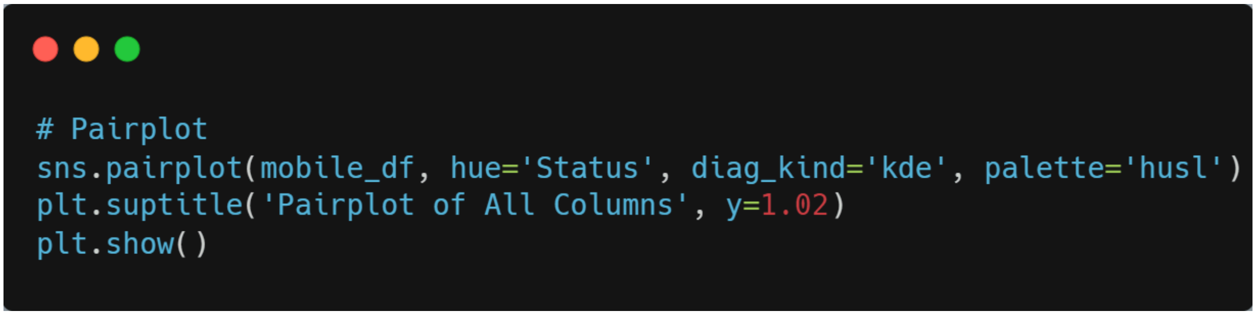


```
# Label encoding for categorical columns
label_encoder = LabelEncoder()
categorical_cols = ['Color', 'Brand Origin', 'Brand and Model', 'Status']
for col in categorical_cols:
    mobile_df[col] = label_encoder.fit_transform(mobile_df[col])


# Remove outliers based on Price(Toman)
Q1 = mobile_df['Price(Toman)'].quantile(0.25)
Q3 = mobile_df['Price(Toman)'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
mobile_df = mobile_df[(mobile_df['Price(Toman)'] >= lower_bound) & (mobile_df['Price(Toman)'] <= upper_bound)]




# Split into X and y
X = mobile_df.drop('Price(Toman)', axis=1)
y = mobile_df['Price(Toman)']


# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=519)


# Train Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)


# Predictions
y_pred = model.predict(X_test)


# Evaluate model
mse = mean_squared_error(y_test, y_pred)


print(f"Mean Squared Error: {mse}")
Convert categorical data into numerical format using Label encoding.
The full explanation is in the comment.
Our test_size is 0.1 and our random_state is 519.
```



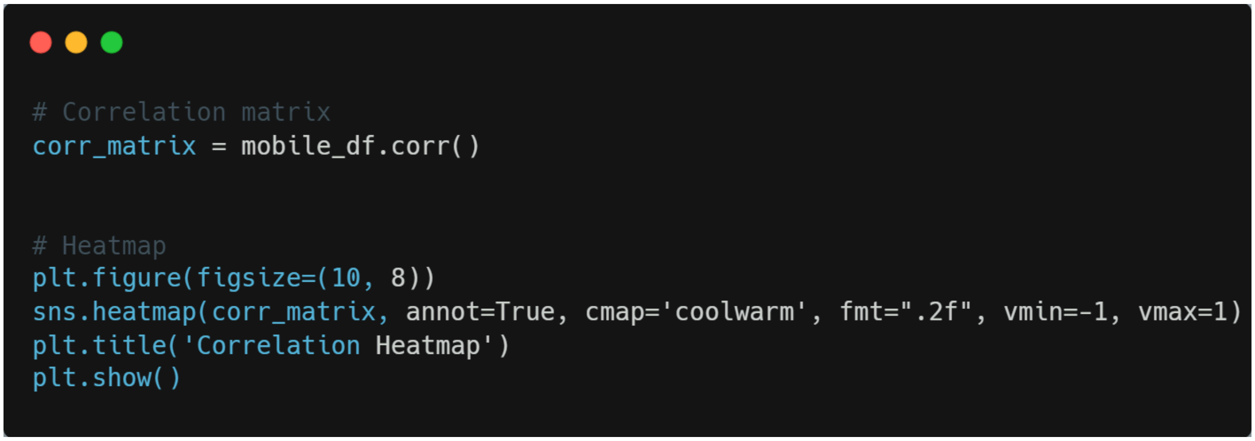
```
# Scatter plot with fitted line
plt.scatter(y_test, y_pred, color='blue', label='Actual vs. Predicted')
plt.plot(y_test, y_test, color='red', linewidth=2, label='Fitted Line')


plt.title('Actual vs. Predicted Prices')
plt.xlabel('Actual Price')
plt.ylabel('Predicted Price')
plt.legend()
plt.show()
```

Show the Actual vs. Predicted Prices.



```
# Pairplot
sns.pairplot(mobile_df, hue='Status', diag_kind='kde', palette='husl')
plt.suptitle('Pairplot of All Columns', y=1.02)
plt.show()
```

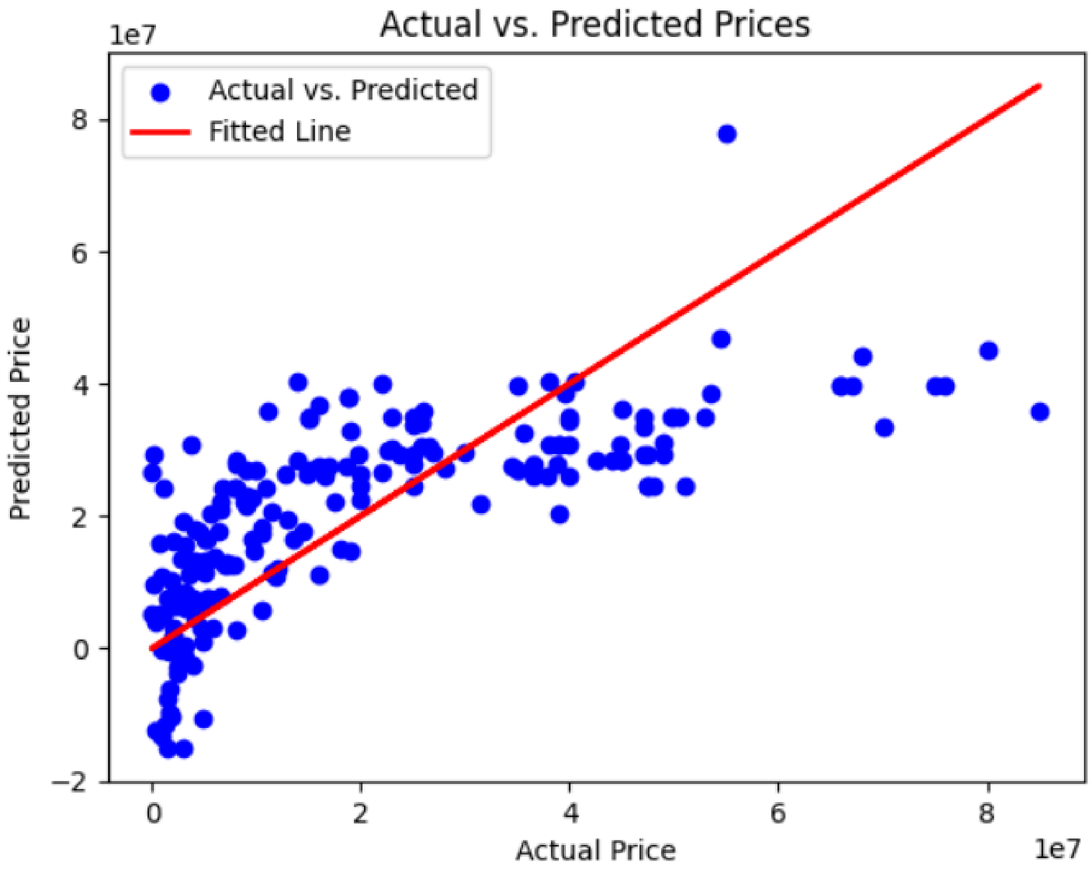
Plot Pairplot for all columns.
A pairplot visually explores relationships and distributions between variables, aiding in validating assumptions and guiding the linear regression modeling process. It helps identify outliers, assess linearity, and detect multicollinearity among predictors.

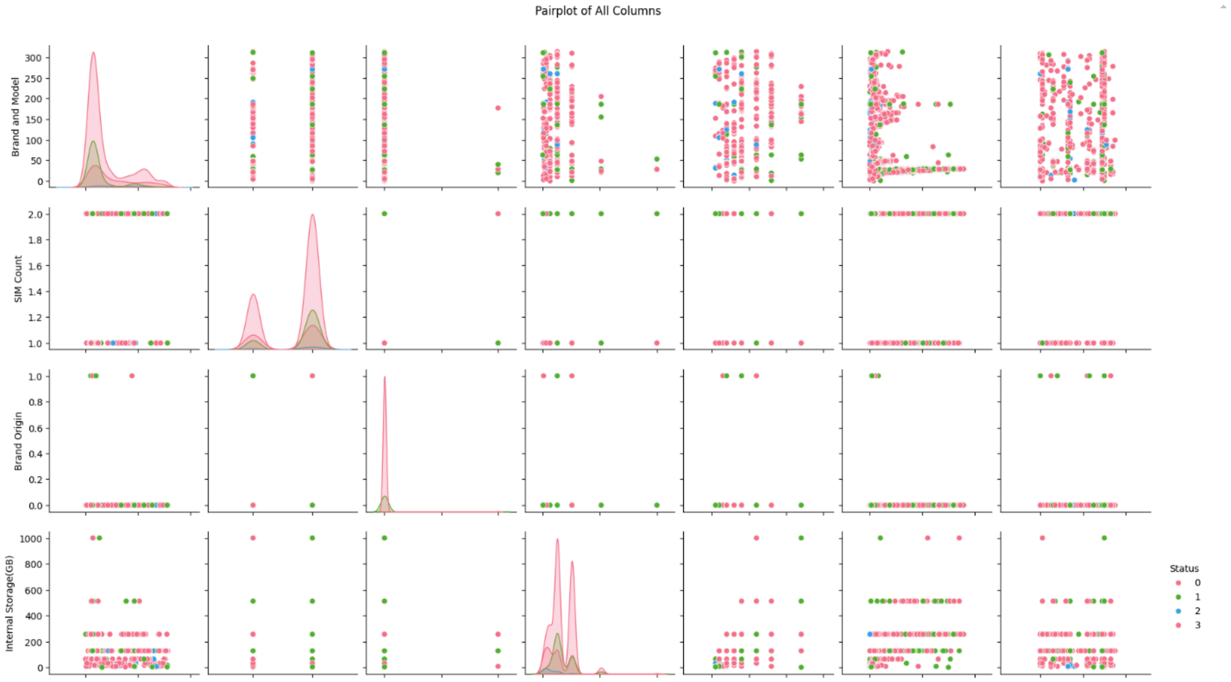


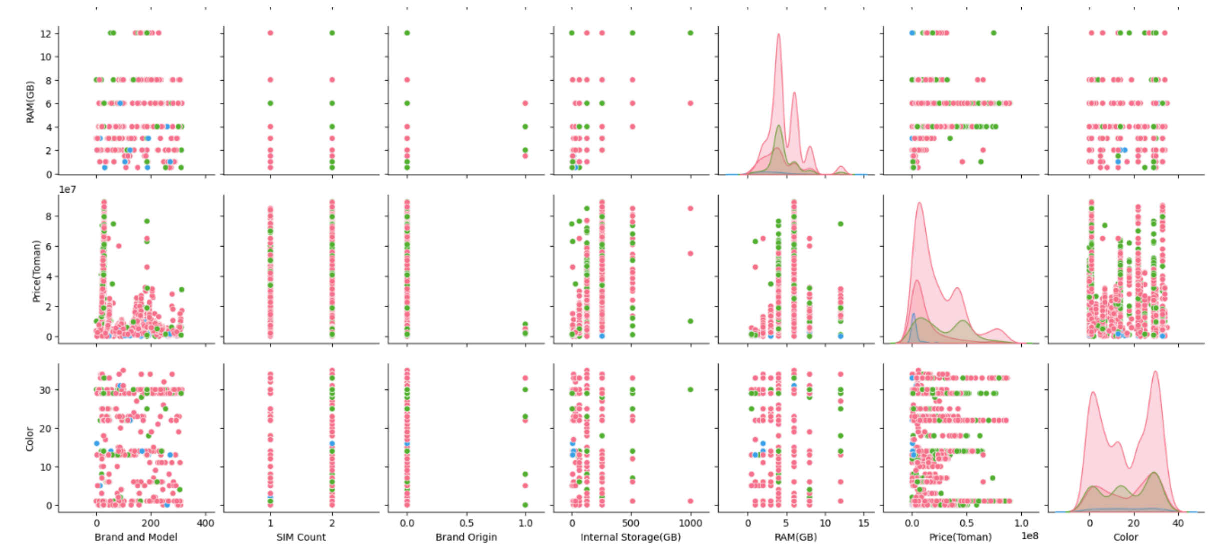
```
# Correlation matrix
corr_matrix = mobile_df.corr()


# Heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f", vmin=-1, vmax=1)
plt.title('Correlation Heatmap')
plt.show()
```

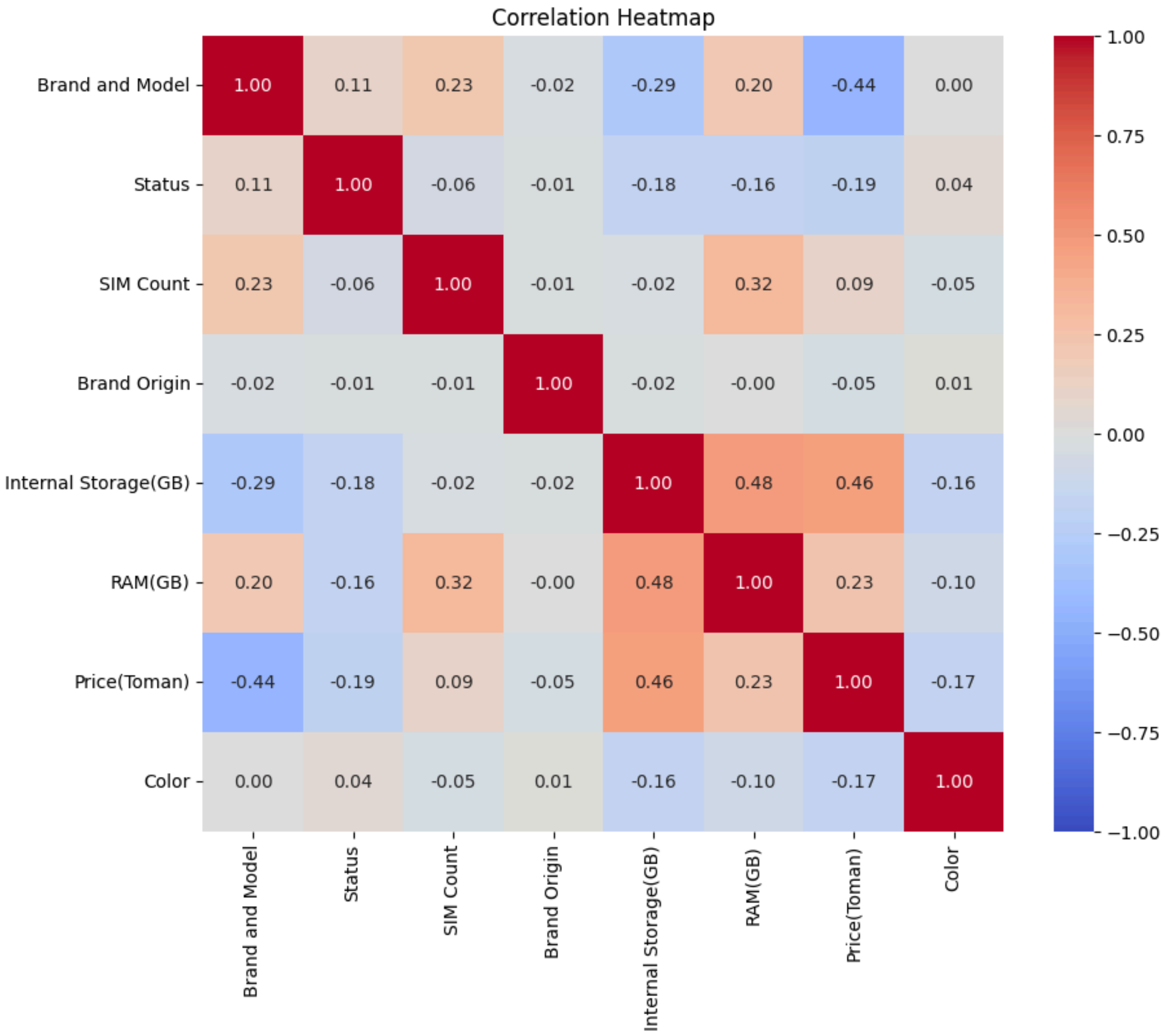
Here we plot correlation heatmap.
A correlation heatmap summarizes pairwise correlations between variables in a dataset using color intensity, making it easy to identify strong and weak relationships at a glance, aiding in feature selection and multicollinearity assessment for modeling.

```
Mean Squared Error: 186517807945521.72
```



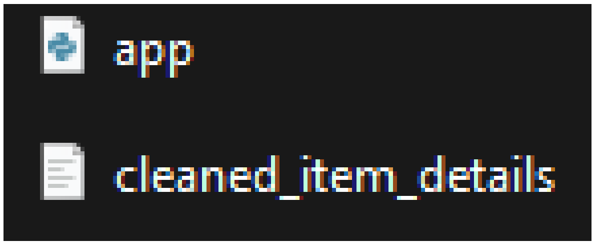



RAM(GB)
Price(Toman)
1e7
Color
Brand and Model
SIM Count
Brand Origin
Internal Storage(GB)
RAM(GB)
Price(Toman)
1e8
Color


Correlation Heatmap
Brand and Model 1.00 0.11 0.23 -0.02 -0.29 0.20 -0.44 0.00
Status 0.11 1.00 -0.06 -0.01 -0.18 -0.16 -0.19 0.04
SIM Count 0.23 -0.06 1.00 -0.01 -0.02 0.32 0.09 -0.05
Brand Origin -0.02 -0.01 -0.01 1.00 -0.02 -0.00 -0.05 0.01
Internal Storage(GB) -0.29 -0.18 -0.02 -0.02 1.00 0.48 0.46 -0.16
RAM(GB) 0.20 -0.16 0.32 -0.00 0.48 1.00 0.23 -0.10
Price(Toman) -0.44 -0.19 0.09 -0.05 0.46 0.23 1.00 -0.17
Color 0.00 0.04 -0.05 0.01 -0.16 -0.10 -0.17 1.00
Brand and Model
Status
SIM Count
Brand Origin
Internal Storage(GB)
RAM(GB)
Price(Toman)
Color
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

3.1 Price Prediction:

This folder Contains:

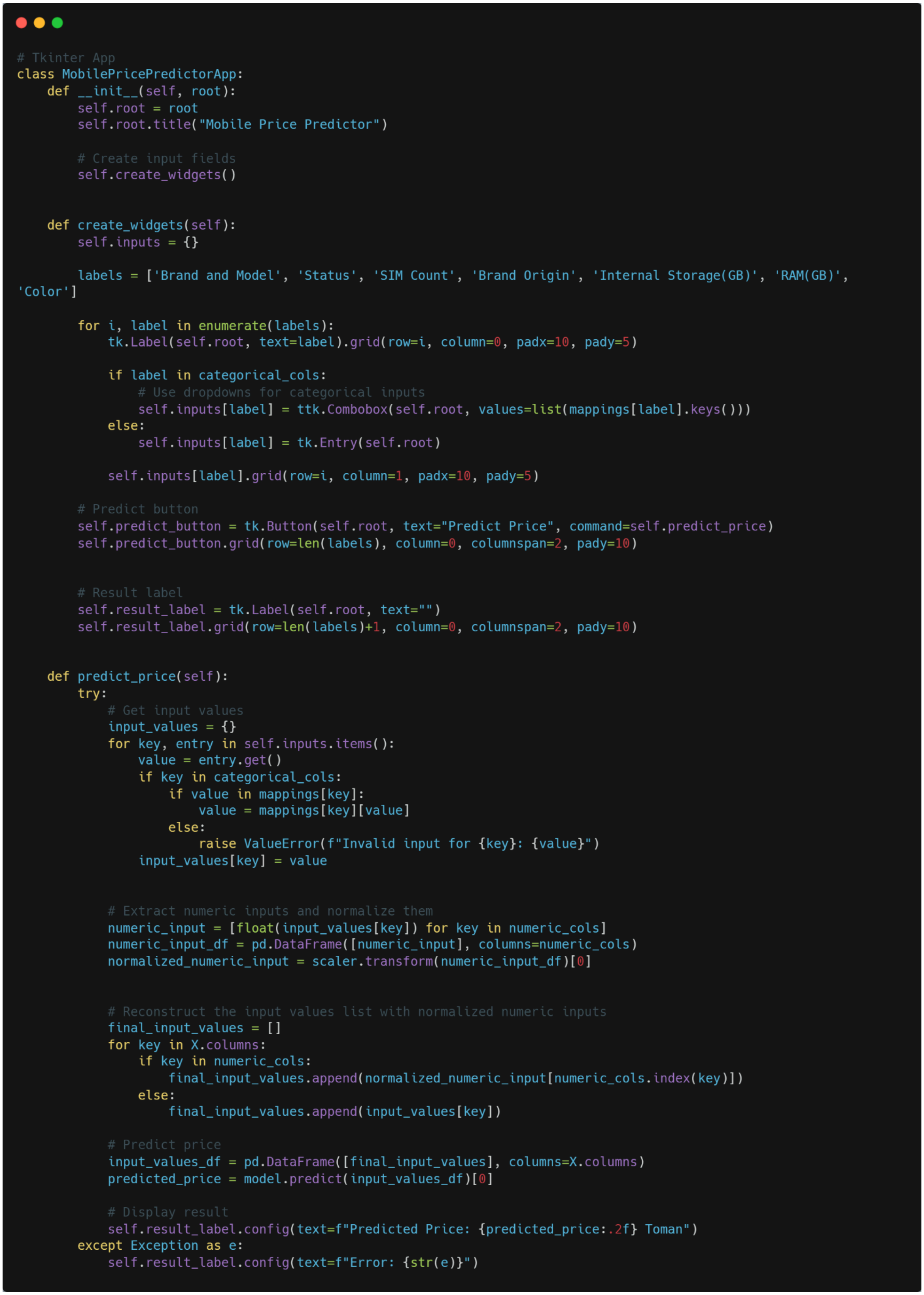


app

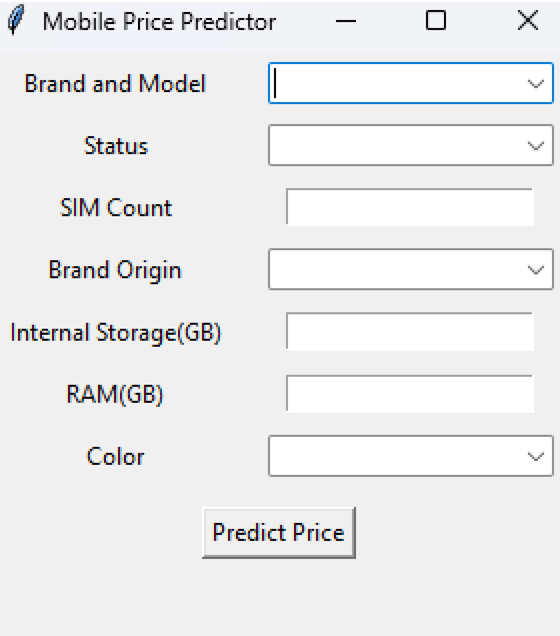
cleaned_item_details

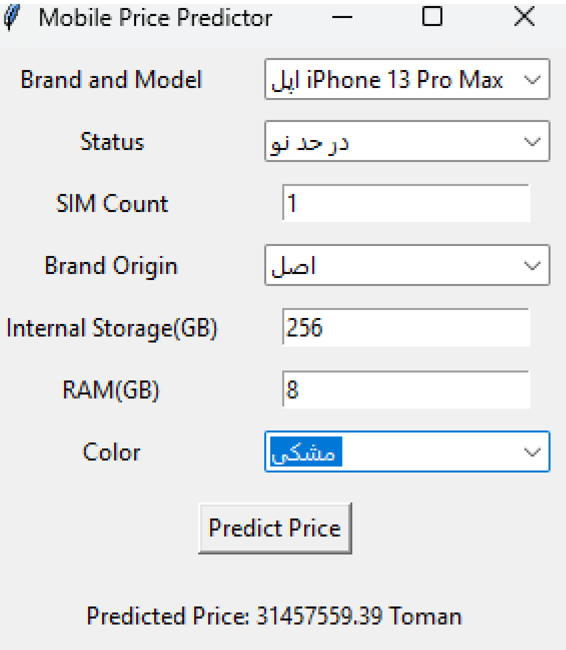
cleaned_item_details.csv is our new dataset after cleaning.
app.py is the application that predicts the mobile price based on the features input by the user.

We use the model we created in the last part to build an application using the tkinter library.



```python
# Tkinter App
class MobilePricePredictorApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Mobile Price Predictor")

        # Create input fields
        self.create_widgets()


    def create_widgets(self):
        self.inputs = {}

        labels = ['Brand and Model', 'Status', 'SIM Count', 'Brand Origin', 'Internal Storage(GB)', 'RAM(GB)',
'Color']

        for i, label in enumerate(labels):
            tk.Label(self.root, text=label).grid(row=i, column=0, padx=10, pady=5)

            if label in categorical_cols:
                # Use dropdowns for categorical inputs
                self.inputs[label] = ttk.Combobox(self.root, values=list(mappings[label].keys()))
            else:
                self.inputs[label] = tk.Entry(self.root)

            self.inputs[label].grid(row=i, column=1, padx=10, pady=5)

        # Predict button
        self.predict_button = tk.Button(self.root, text="Predict Price", command=self.predict_price)
        self.predict_button.grid(row=len(labels), column=0, columnspan=2, pady=10)


        # Result label
        self.result_label = tk.Label(self.root, text="")
        self.result_label.grid(row=len(labels)+1, column=0, columnspan=2, pady=10)


    def predict_price(self):
        try:
            # Get input values
            input_values = {}
            for key, entry in self.inputs.items():
                value = entry.get()
                if key in categorical_cols:
                    if value in mappings[key]:
                        value = mappings[key][value]
                    else:
                        raise ValueError(f"Invalid input for {key}: {value}")
                input_values[key] = value


            # Extract numeric inputs and normalize them
            numeric_input = [float(input_values[key]) for key in numeric_cols]
            numeric_input_df = pd.DataFrame([numeric_input], columns=numeric_cols)
            normalized_numeric_input = scaler.transform(numeric_input_df)[0]


            # Reconstruct the input values list with normalized numeric inputs
            final_input_values = []
            for key in X.columns:
                if key in numeric_cols:
                    final_input_values.append(normalized_numeric_input[numeric_cols.index(key)])
                else:
                    final_input_values.append(input_values[key])

            # Predict price
            input_values_df = pd.DataFrame([final_input_values], columns=X.columns)
            predicted_price = model.predict(input_values_df)[0]

            # Display result
            self.result_label.config(text=f"Predicted Price: {predicted_price:.2f} Toman")
        except Exception as e:
            self.result_label.config(text=f"Error: {str(e)}")
```





Brand and Model: اپل iPhone 13 Pro Max

Status: در حد نو

SIM Count: 1

Brand Origin: اصل

Internal Storage(GB): 256

RAM(GB): 8

Color: مشکی

Predicted Price: 31457559.39 Toman