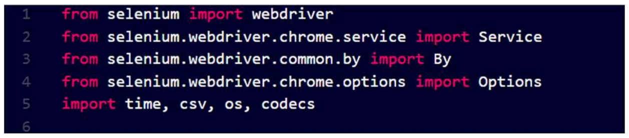
# *Cheat Sheet*

👋Hi.. I am here to share how I made this **Blinkit product scraping** project using Python and Selenium. This guide explains the purpose, logic, and tools used at each step to help you understand not just what the code does, but why it works the way it does.

Here's the Complete Breakdown:

## 1. Imports:

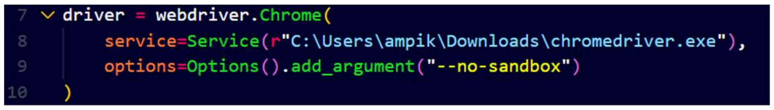


```python
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.options import Options
import time, csv, os, codecs
```

- selenium: Used for web automation and scraping.
- time: Allows waiting (sleep) to let pages load.
- csv: Used for writing the extracted data to a CSV file.
- os: Used to determine the desktop path dynamically.
- codes: Ensures the file is written with UTF-8 BOM for compatibility (especially with Excel).

---

## 2. Setting Up WebDriver

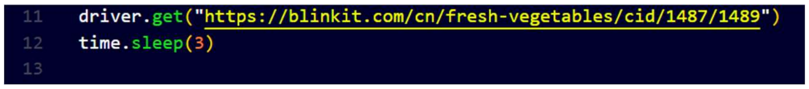


```python
driver = webdriver.Chrome(
    service=Service(r"C:\Users\ampik\Downloads\chromedriver.exe"),
    options=Options().add_argument("--no-sandbox")
)
```

- Launches a Chrome browser instance using Selenium.
- chromedriver.exe: Driver binary path (must match your Chrome version).
- --no-sandbox: (Generally used in headless environments; mostly safe to omit locally).

---

## 3. Navigating to the Target Page



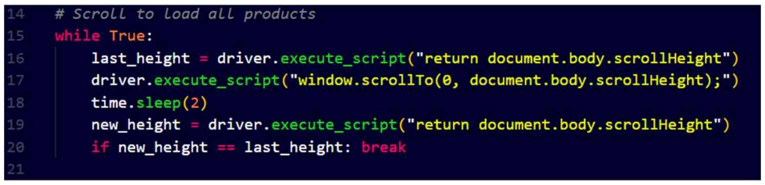
```python
driver.get("https://blinkit.com/cn/fresh-vegetables/cid/1487/1489")
time.sleep(3)
```

Opens the Blinkit page for fresh vegetables.

- time.sleep(3): Waits 3 seconds for the page to load.

---

## 4. Scrolling to Load All Products

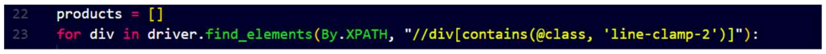


```python
# Scroll to load all products
while True:
    last_height = driver.execute_script("return document.body.scrollHeight")
    driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")
    time.sleep(2)
    new_height = driver.execute_script("return document.body.scrollHeight")
    if new_height == last_height: break
```

- This loop scrolls down to the bottom of the page repeatedly.
- It waits for new content to load, and checks if the height of the page changed.
- If no new content appears, the loop stops (all products loaded).

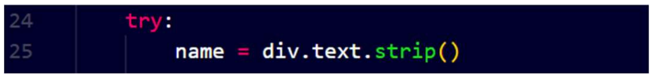
---

## 5. Extract product details



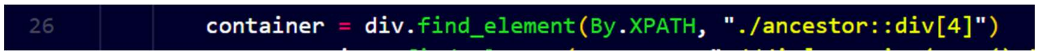
```python
products = []
for div in driver.find_elements(By.XPATH, "//div[contains(@class, 'line-clamp-2')]"):
```

- Finds all elements that contain product names (based on a specific class).
- Loops over each product name div.



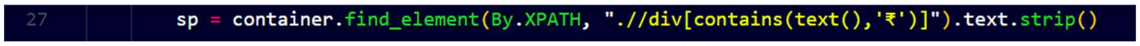
```python
    try:
        name = div.text.strip()
```

- Extracts and cleans the product name.



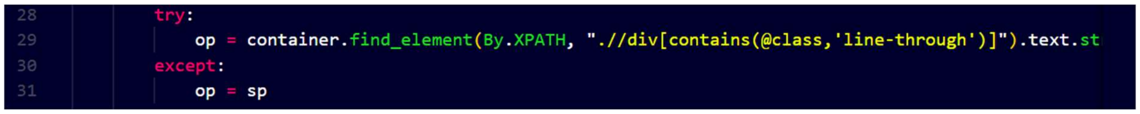
```python
        container = div.find_element(By.XPATH, "./ancestor::div[4]")
```

- Goes up 4 levels from the name element to reach the main product container.



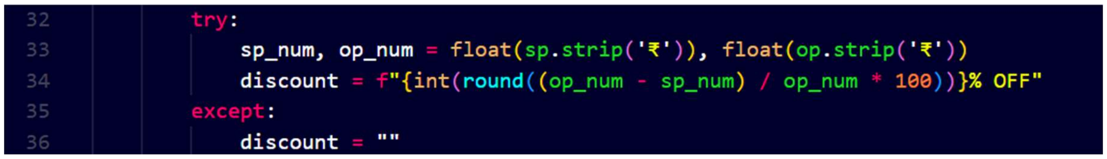
```python
        sp = container.find_element(By.XPATH, ".//div[contains(text(),'₹')]").text.strip()
```

- Finds the selling price (₹ price without strike-through).



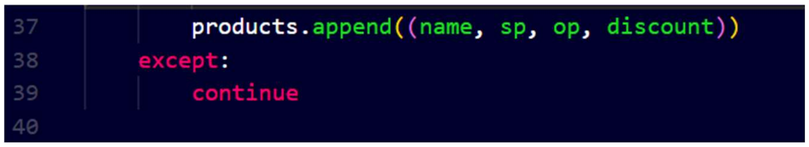
```python
        try:
            op = container.find_element(By.XPATH, ".//div[contains(@class,'line-through')]").text.st
        except:
            op = sp
```

- Try to find the original price (strike-through price).
- If not found, assumes the original price = selling price (no discount).



```python
        try:
            sp_num, op_num = float(sp.strip('₹')), float(op.strip('₹'))
            discount = f"{int(round((op_num - sp_num) / op_num * 100))}% OFF"
        except:
            discount = ""
```

- Try to convert prices to float and calculate discount percentage.
- If calculation fails (bad data), discount is set to empty string.



```python
        products.append((name, sp, op, discount))
    except:
        continue
```

- Adds the product tuple to the products list.

- If any error occurs in the block, it skips the product.

---

### 6. Close the browser

```python
driver.quit()
```

- Closes the Chrome browser after scraping is complete.

---

### 7. Save products to CSV file

```python
path = os.path.join(os.path.expanduser("~"), "Desktop", "blinkit_products.csv")
```

- Creates the path for the output CSV file on the user's Desktop.

```python
with codecs.open(path, "w", encoding="utf-8-sig") as f:
    csv.writer(f).writerows([["Name", "Selling Price", "Original Price", "Discount"]] + products)
```

- Opens the CSV file with utf-8-sig encoding (help with special characters in Excel).
- Writes a header row + all product rows to the file.

### 8. Confirmation

```python
print(f"✅ {len(products)} products saved to {path}")
```

- Prints a message showing how many products were saved and where.

## Why I Used Selenium?

I used Selenium in my script for these main reasons:

**1. The website uses JavaScript to load content**

- The Blinkit website does not show all product data immediately in HTML.
- It loads products dynamically using JavaScript after the page opens.
- Normal tools like requests or BeautifulSoup cannot see this dynamic content.
- Selenium works by controlling a real browser, so it can see and interact with the full page after JavaScript runs.

**2. I need to scroll to load more products**

- Blinkit uses infinite scroll to show more items when you reach the bottom of the page.
- Selenium can scroll the page automatically, just like a real user.
- This helps in loading all the products before extracting the data.

**3. I need to extract data from the fully loaded page**

- Once all products are loaded, you use Selenium to:

- Find product names
- Get prices
- Get discount details

These elements only appear after the page finishes loading and scrolling, which Selenium handles well. If the page had all the data in the HTML (without JavaScript), you wouldn't need Selenium, but because this site is dynamic, Selenium is the right tool.

## Why Is XPath Needed?

**1. Precise element targeting**

The page has many similar-looking elements. XPath helps you select only the correct ones, like product titles or prices.

**2. Flexible navigation of HTML**

XPath lets you move up and down the DOM (Document Object Model), so you can find related elements for example, getting a price from a parent or sibling element of the product name.

**3. No reliable IDs or classes**

Some websites don't give unique IDs or consistent class names. XPath allows you to find elements based on structure, position, or text content.

**4. Dynamic content**

Since the content is loaded dynamically, simple selectors (like tag name or class) might not always work correctly. XPath provides more control and accuracy.

I need XPath because it allows accurate and flexible selection of elements in a complex, dynamically loaded webpage. It's especially useful when scraping nested or related data, as in your Blinkit scraping project.

## Summary of This Project

This project is built using Python and Selenium to automate the extraction of product information from the Blinkit website's fresh vegetables section. Since the site dynamically loads content through JavaScript and uses infinite scrolling, Selenium is used to control a real Chrome browser that scrolls through the page, waits for elements to load, and collects complete product listings. The script extracts each product's name, selling price, original price, and calculates the discount percentage using XPath to navigate the HTML structure. The data is then stored in a structured CSV format. From a data analyst's perspective, this project provides a reliable method to collect real-time pricing data that can be used for competitor price tracking, market trend analysis, demand forecasting, and identifying discount patterns. It automates a previous manual process, enabling analysts to work with clean, up-to-date data for reporting, visualization, or integration into larger data workflows.