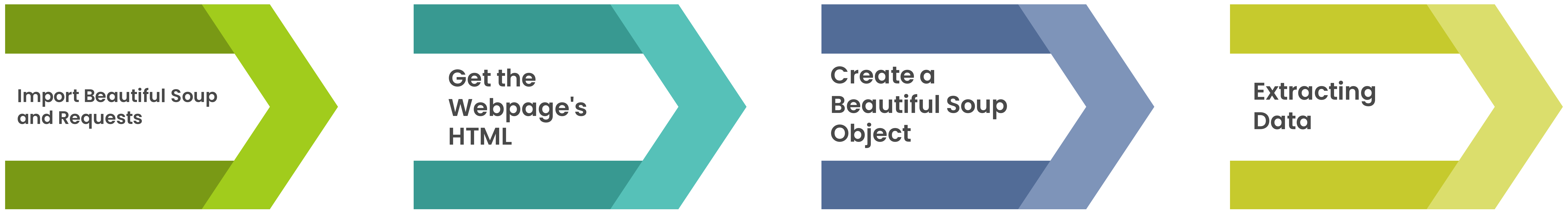
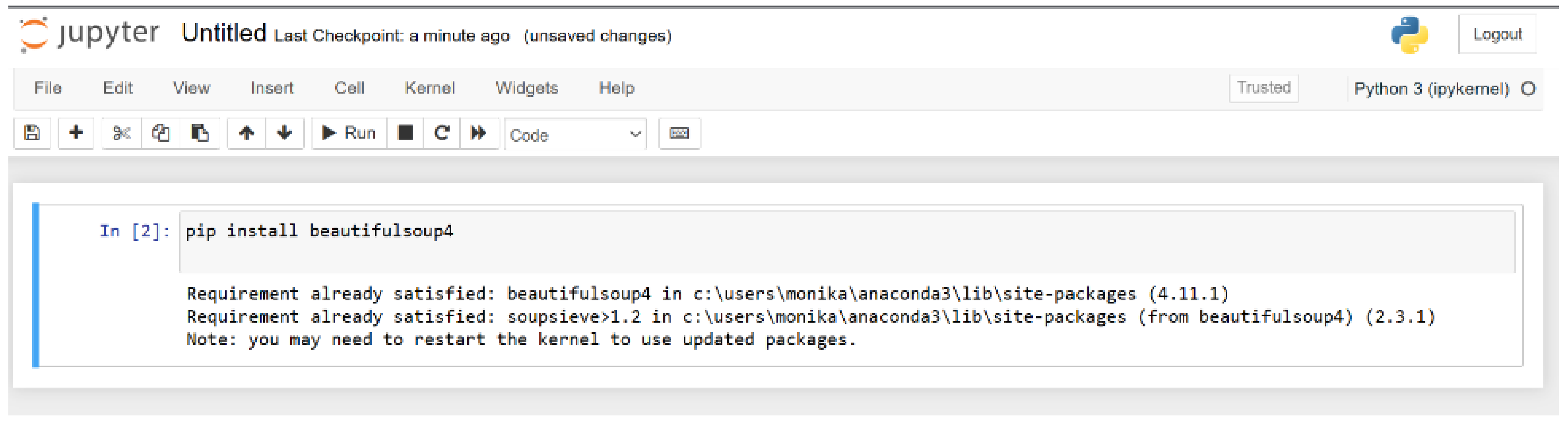
Beautiful Soup

Beautiful Soup is a Python library that is used for web scraping purposes. It allows you to parse HTML or XML documents, extract data, and navigate the document structure. You can use it to extract specific information from web pages by searching for HTML elements and their attributes.


Import Beautiful Soup and Requests
Get the Webpage's HTML
Create a Beautiful Soup Object
Extracting Data

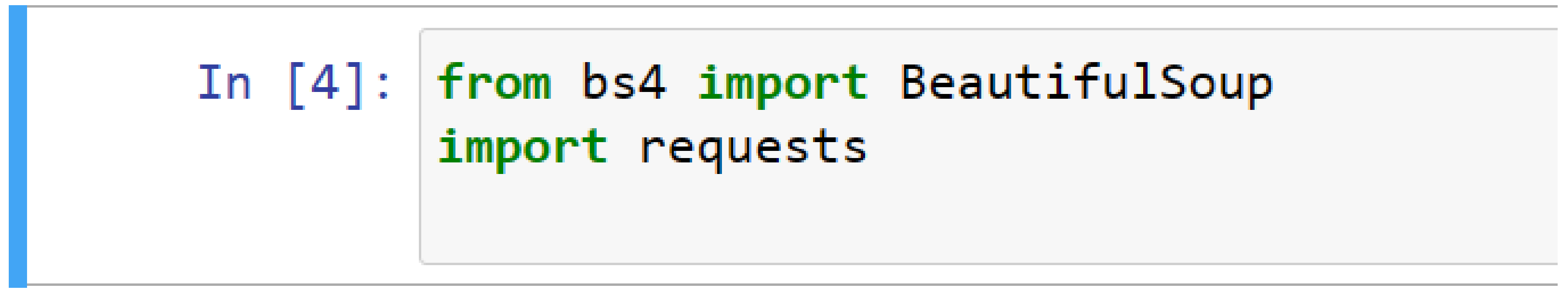
Step 1- Installation

pip install beautifulsoup4


jupyter Untitled Last Checkpoint: a minute ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code

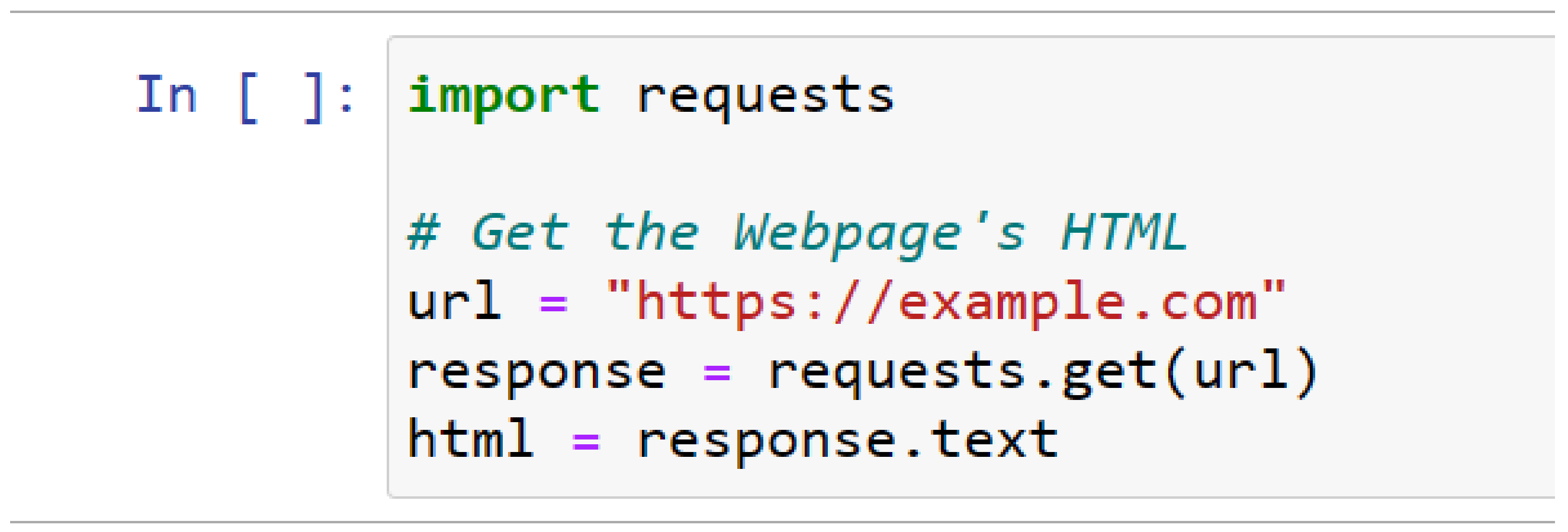
```
In [2]: pip install beautifulsoup4

        Requirement already satisfied: beautifulsoup4 in c:\users\monika\anaconda3\lib\site-packages (4.11.1)
        Requirement already satisfied: soupsieve>1.2 in c:\users\monika\anaconda3\lib\site-packages (from beautifulsoup4) (2.3.1)
        Note: you may need to restart the kernel to use updated packages.
```

Step 2-Import Beautiful Soup and Requests



```
In [4]: from bs4 import BeautifulSoup
        import requests
```

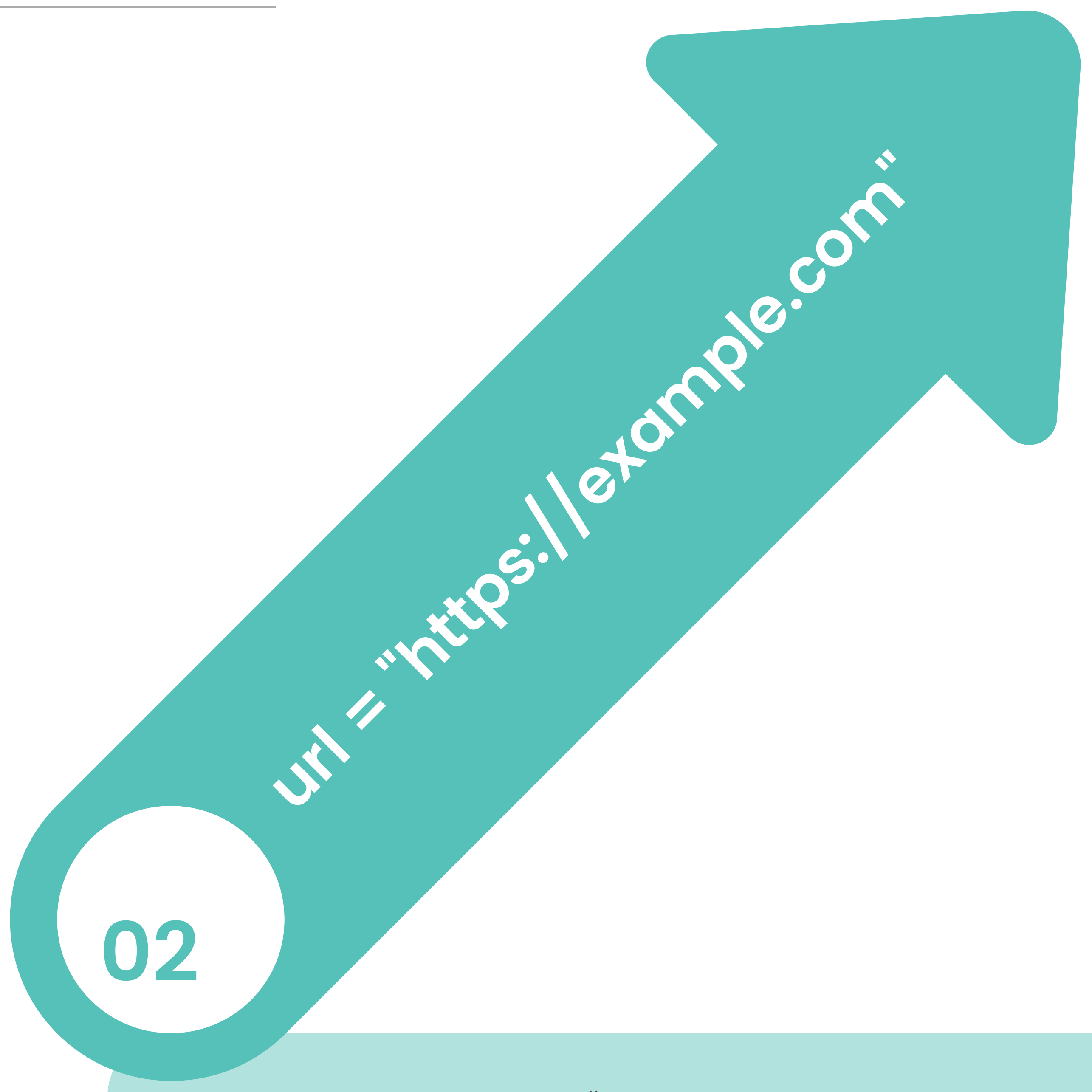
Step 3 →Get the Webpage's HTML



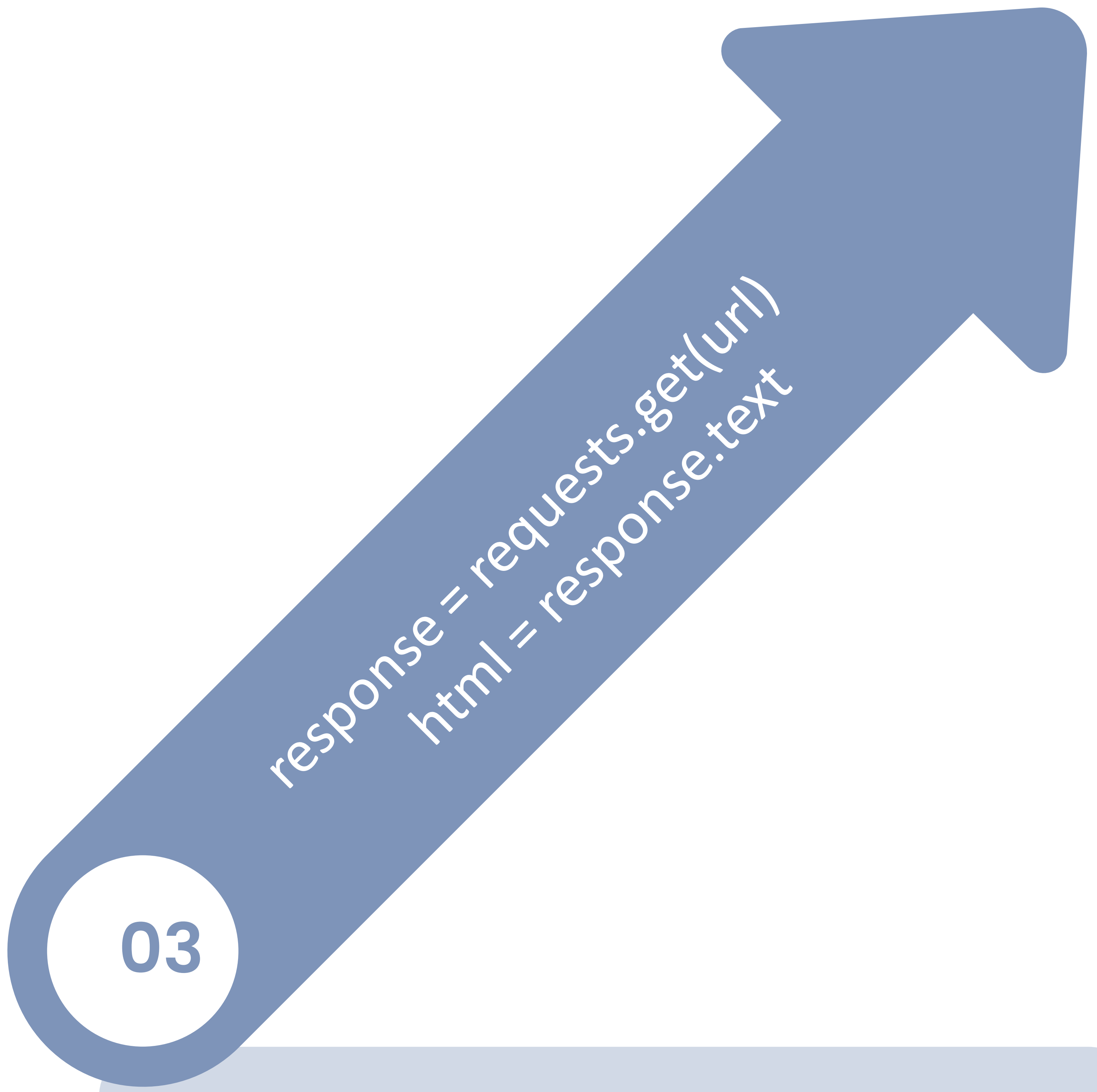
```
In [ ]: import requests

        # Get the Webpage's HTML
        url = "https://example.com"
        response = requests.get(url)
        html = response.text
```

Replace "https://example.com" with the *actual URL of the webpage you want to scrape.*

.



Use the requests.get() method to send an HTTP GET request to the specified URL. This method is part of the requests library in Python, and it allows you to retrieve the content of a webpage by making an HTTP GET request to the URL.



The response from the server is then stored in the response variable, and you can access the HTML content of the page using response.text. This is a common approach when performing web scraping or when you need to interact with web resources to retrieve data.

Step 4 → Object creation

```
soup = BeautifulSoup(html, 'html.parser')
```

- ***Creating a Beautiful Soup object is essential when you're working with web scraping or parsing HTML documents for several reasons:***

Navigational Tools: Beautiful Soup provides various methods and functions for traversing the HTML tree, allowing you to move up, down, and sideways within the document to locate the elements you're interested in. This flexibility is essential for data scraping.

Support for Different Parsers: Beautiful Soup supports multiple parsers (e.g., 'html.parser', 'lxml', 'html5lib'), allowing you to choose the most suitable one for your specific task or to handle different HTML formats.

Integration with Requests: When combined with the requests library, Beautiful Soup allows you to fetch web pages, parse them, and extract data in a seamless workflow.

- **Parse HTML and XML**: HTML and XML documents are often complex, and manually parsing them in raw text form can be. Beautiful Soup allows to create a structured parse tree from the document.

- **Data Sanitization**: When scraping data, it's common to encounter unexpected characters or malicious code. Beautiful Soup can help you sanitize the data by removing or escaping problematic content.

Facilitates Data Extraction Rules: When scraping data, you often need to define rules for extracting specific elements or content. Beautiful Soup provides a way to define these rules and apply them consistently across multiple web pages.

Step 5: Extracting Data

Now that you have a Beautiful Soup object, you can use it to navigate the HTML and extract data. Here are a few common operations:

Getting the page title:

python

```python
title = soup.title.string
print("Title of the page:", title)
```

Finding a specific tag:

Suppose you want to extract the first paragraph (<p>) on the page:

python

```python
paragraph = soup.find('p')
print("First paragraph:", paragraph.text)
```

Finding all instances of a tag:

To extract all the links (<a>) on the page:

python

```python
links = soup.find_all('a')
for link in links:
    print("Link:", link['href'])
```

Navigating the document tree:

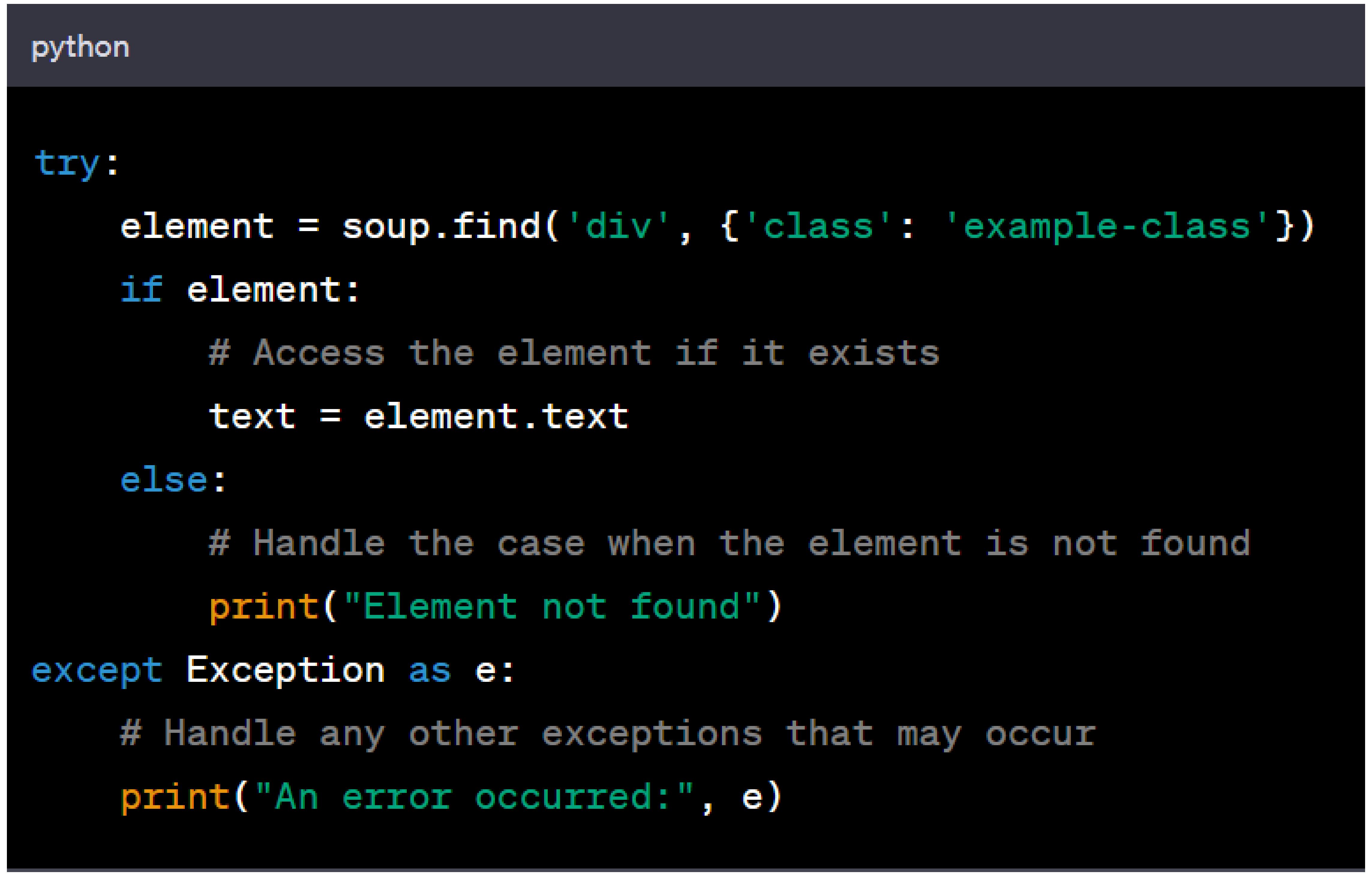
You can navigate up and down the document tree. For example, to find the parent of a tag:

python

```python
parent = paragraph.parent
print("Parent of the first paragraph:", parent.name)
```

Step 6: Error Handling

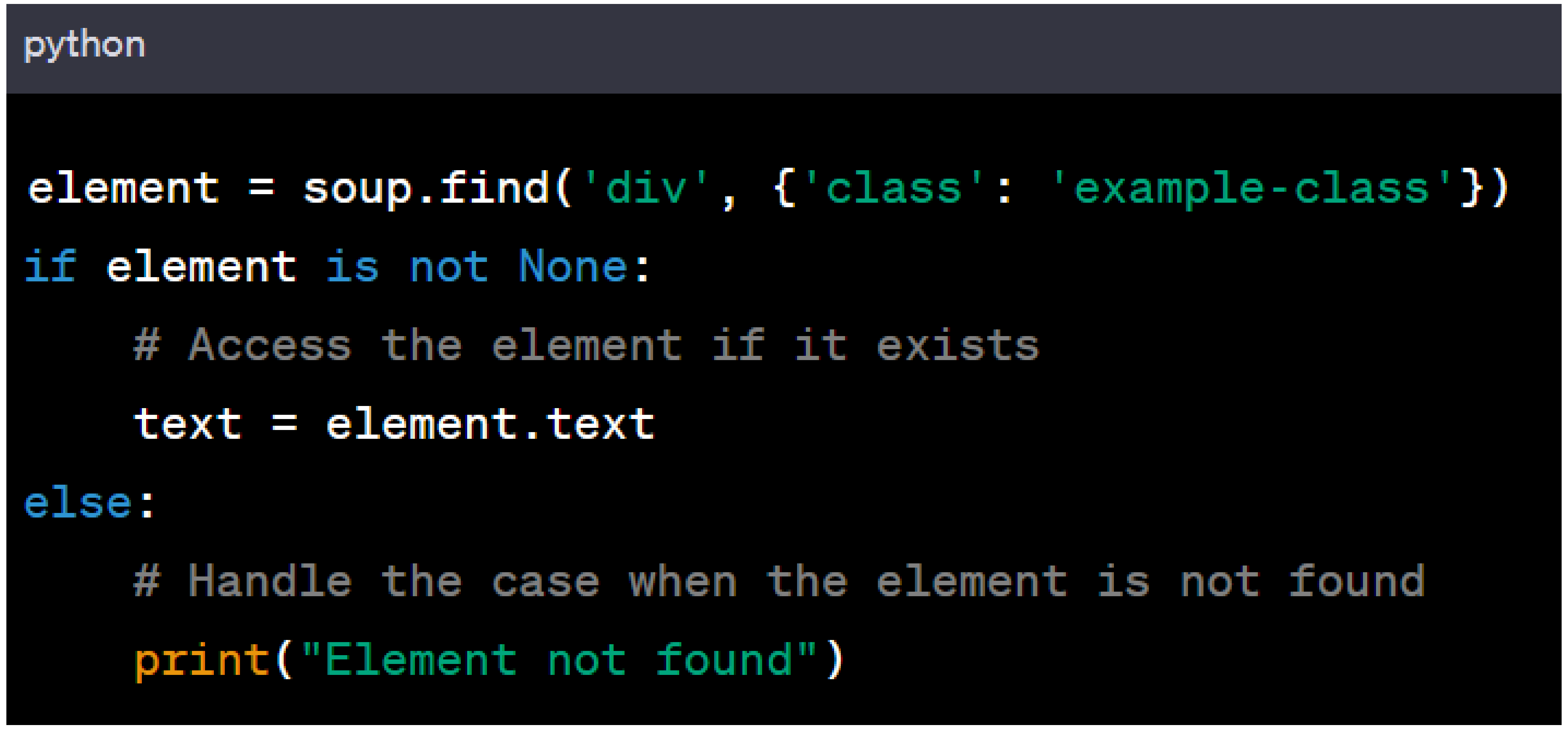
When working with web scraping, you should be prepared for exceptions and handle them appropriately. For example, check if an element exists before trying to access it.
Here's how you can handle errors and check if an element exists before trying to access it:



```python
try:
    element = soup.find('div', {'class': 'example-class'})
    if element:
        # Access the element if it exists
        text = element.text
    else:
        # Handle the case when the element is not found
        print("Element not found")
except Exception as e:
    # Handle any other exceptions that may occur
    print("An error occurred:", e)
```

Check for None:

You can also check if an element exists by comparing it to None. If Beautiful Soup doesn't find the element, it returns None, so you can check for its presence explicitly:



python

```python
element = soup.find('div', {'class': 'example-class'})
if element is not None:
    # Access the element if it exists
    text = element.text
else:
    # Handle the case when the element is not found
    print("Element not found")
```

Step 7: Data Processing and Storage

Data processing involves, the following steps.....

Data Cleaning: The data you extract may require cleaning to remove unnecessary characters, format it correctly, or handle missing values. You can use Python's string manipulation and data cleaning functions to do this.

Data Transformation: You might need to transform the data into a different structure or format. For instance, you could convert dates, currencies, or measurements to a standardized format.

Data Analysis: If your goal is to gain insights from the data, you can perform data analysis using libraries like Pandas, NumPy, or Matplotlib. This allows you to calculate statistics, create visualizations, and draw conclusions from the extracted data.

Natural Language Processing (NLP): If the data includes text, you can use NLP libraries like NLTK or spaCy to analyze and extract information from the text. This is useful for sentiment analysis, topic modeling, or text summarization.

Data storage:

CSV Files: Storing data in CSV (Comma-Separated Values) format is a common method. You can use Python's csv module to write data to CSV files. This is suitable for structured data that can be easily represented in rows and columns.

JSON Files: If the data has a more complex structure, JSON (JavaScript Object Notation) format may be a better choice. Python's json module allows you to work with JSON files.

Databases: For more extensive data or when you need to query and retrieve data efficiently, consider using a database. Python has libraries like SQLite, MySQL, and PostgreSQL that allow you to store data in relational databases. SQLAlchemy is a popular ORM (Object-Relational Mapping) library for working with databases.

Cloud Storage: You can store data in cloud-based storage Data Frames: If you're working with structured data and plan to perform data analysis, Pandas DataFrames provide a convenient way to store and manipulate data within your Python script.