Collecting Data in Python

Course:
INFO-6145 Data Science and Machine Learning

FANSHAWE

Revised by:
Mohammad Noorchenarboo

November 28, 2024

Contents

Current Section

Introduction to HTTP Methods

HTTP methods define the actions that can be performed on resources in a web service. The most common methods are:

- **GET:** Retrieve data.
- **POST:** Send data to create a new resource.
- **PUT:** Update an existing resource.
- **DELETE:** Remove a resource.

Why Are HTTP Methods Important?

- Provide a standardized way to interact with resources in a web service.
- Simplify communication between client and server.
- Help organize and implement RESTful APIs efficiently.

GET Method: Retrieving Data

The **GET** method is used to retrieve data from the server.

Characteristics of GET

- Requests data from a specific resource.
- Does not modify the data on the server.
- Parameters are sent in the URL (query string).

GET Method: Retrieving Data

Example: GET Request

Client Request:

```
GET /books/1 HTTP/1.1
Host: api.example.com
```

Server Response:

```
{
  "id": 1,
  "title": "Introduction to Python",
  "author": "John Doe",
  "published": 2022}
```

POST Method: Creating a Resource

The **POST** method is used to create new resources on the server.

Characteristics of POST

- Sends data to the server in the request body.
- Often used to submit forms or upload files.
- Can result in creating a new resource.

POST Method: Creating a Resource

Example: POST Request

Client Request:

```
POST /books HTTP/1.1
Host: api.example.com
Content-Type: application/json

{
  "title": "Advanced Python",
  "author": "Jane Smith",
  "published": 2023
}
```

Server Response:

```
{
  "id": 2,
  "message": "Book created successfully."
}
```

PUT Method: Updating a Resource

The **PUT** method is used to update an existing resource on the server.

Characteristics of PUT

- Sends data to update a resource identified by the URL.
- The entire resource is usually replaced.
- Often used for editing or modifying data.

PUT Method: Updating a Resource

Example: PUT Request

Client Request:

```
PUT /books/1 HTTP/1.1
Host: api.example.com
Content-Type: application/json

{
  "title": "Introduction to Python - Updated",
  "author": "John Doe",
  "published": 2022
}
```

Server Response:

```
{
  "id": 1,
  "message": "Book updated successfully."
}
```

DELETE Method: Removing a Resource

The **DELETE** method is used to remove a resource from the server.

Characteristics of DELETE

- Deletes the resource identified by the URL.
- Does not send a body in the request.
- Often used to remove data from the server.

DELETE Method: Removing a Resource

Example: DELETE Request

Client Request:

```
DELETE /books/1 HTTP/1.1
Host: api.example.com
```

Server Response:

```
{
  "id": 1,
  "message": "Book deleted successfully."
}
```

Current Section

Using Python for HTTP Requests

Python provides libraries to interact with web services, such as ‘requests‘. This library in Python is a popular HTTP library that simplifies making HTTP requests. It provides easy-to-use interfaces for handling responses, cookies, and authentication.

Advantages of the ‘requests‘ Library

- Easy to use for making HTTP requests.
- Supports various methods like GET, POST.
- Handles common tasks like cookies, headers, and sessions.
- Simplifies handling of JSON data and file uploads.
- Provides robust error handling and response status checking.

Using Python for HTTP Requests

Example of a GET Request

```
import requests
# Example GET request
response = requests.get('https://api.example.com/data')
if response.status_code == 200:
    print(response.json())
```

Example of a POST Request

```
import requests
# Example POST request
data = {'key': 'value'}
response = requests.post('https://api.example.com/data',
    json=data)
if response.status_code == 201:
    print('Data successfully sent!')
```

Best Practices for HTTP Requests

Warnings for HTTP Requests

- Always handle exceptions to avoid crashes.
- Ensure sensitive data is encrypted when sending over POST.

Example of Exception Handling

```python
try:
    response = requests.get('https://api.example.com/data')
    response.raise_for_status()  # Raise HTTPError for bad
        responses
    print(response.json())
except requests.exceptions.RequestException as e:
    print(f'Error occurred: {e}')
```

Current Section

Receiving HTTP Responses

HTTP responses are returned by a server to indicate the status of a client's request. They include a status code and a short message describing the outcome of the request.

Common HTTP Response Codes

Code	Description
200	OK: Request was successful.
201	Created: Resource was successfully created.
400	Bad Request: The server could not understand the request.
401	Unauthorized: Authentication is required.
403	Forbidden: Access to the resource is not allowed.
404	Not Found: The requested resource could not be found.
500	Internal Server Error: The server encountered an error.
503	Service Unavailable: The server is currently unable to handle the request.

Current Section

Python Libraries for Data Collection and Automation

Python offers a range of libraries for web scraping, API interaction, and automation. These libraries simplify tasks such as data collection, browser automation, and interacting with web services.

Popular Libraries and Their Uses

- **Selenium:** For browser automation and dynamic web scraping.
- **Requests:** For making HTTP requests to interact with web services and APIs.
- **BeautifulSoup:** For parsing HTML and extracting data from static web pages.
- **Scrapy:** For large-scale web scraping and crawling tasks.
- **Pandas:** For data manipulation and analysis after collection.

Selenium: Browser Automation

Selenium is a powerful library for automating web browsers. It is widely used for testing web applications and scraping dynamic content.

Key Features of Selenium

- Supports multiple browsers like Chrome, Firefox, and Edge.
- Allows interaction with web elements such as buttons, forms, and links.
- Handles JavaScript-heavy web pages by rendering content dynamically.
- Enables automation of repetitive browser tasks.

Selenium: Browser Automation

Example: Using Selenium to Access a Web Page

```python
from selenium import webdriver
from selenium.webdriver.common.by import By

# Initialize WebDriver
driver = webdriver.Chrome()

# Open a website
driver.get('https://example.com')

# Find an element and interact with it
search_box = driver.find_element(By.NAME, 'q')
search_box.send_keys('Python Selenium')
search_box.submit()

# Close the browser
driver.quit()
```

When to Use Selenium vs Other Libraries

When to Use Selenium

- When interacting with dynamic web pages that use JavaScript.
- For tasks requiring user actions, such as clicking buttons or filling forms.
- To test web applications under various conditions.

Limitations of Selenium

- Slower than libraries like **Requests** for static data extraction.
- Requires additional setup, such as WebDriver installations.
- May face challenges with CAPTCHA-protected pages.

When to Use Alternatives

- **BeautifulSoup:** For simple HTML parsing and data extraction.
- **Requests:** For API interaction and downloading static content.
- **Scrapy:** For large-scale web scraping and web crawling.