Requests and Web Scraping

Requests

Python's `requests` library is a popular and user-friendly HTTP library used for making HTTP requests to APIs or web servers. It is much more intuitive and concise compared to Python's built-in `urllib` module. Here's a comprehensive guide on using the `requests` library along with some examples.

Installation

First, you need to install the `requests` library. It's not included in the Python standard library, so you need to install it separately using pip:

```
pip install requests
```

Making Requests

The `requests` library supports several HTTP methods like GET, POST, PUT, DELETE, etc. Let's explore some common uses:

1. GET Requests

A GET request is used to retrieve data from a specified resource.

```python
import requests

url = 'http://httpbin.org/get'
response = requests.get(url)

print(response.text)  # prints the response content
```

You can also pass parameters in the URL:

```python
payload = {'key1': 'value1', 'key2': 'value2'}
response = requests.get(url, params=payload)

print(response.url)  # prints the full URL with parameters
```

2. POST Requests

A POST request is used to send data to a server to create/update a resource.

```python
url = 'http://httpbin.org/post'
data = {'key': 'value'}

response = requests.post(url, data=data)

print(response.text)
```

For JSON data, use `json` parameter:

```python
json_data = {'key': 'value'}
response = requests.post(url, json=json_data)

print(response.text)
```

3. PUT Requests

PUT requests are used to send data to a server to create or update a resource.

```python
url = 'http://httpbin.org/put'
data = {'key': 'value'}

response = requests.put(url, data=data)

print(response.text)
```

4. DELETE Requests

DELETE requests are used to delete a resource.

```
url = 'http://httpbin.org/delete'

response = requests.delete(url)

print(response.text)
```

Handling Response

When you make a request, the server responds with a response object. This object contains all the information returned by the server.

```
response = requests.get(url)

# Status Code
print(response.status_code)  # 200, 404, 500, etc.

# Content
print(response.content)  # Binary response content
print(response.text)  # String response content
print(response.json())  # JSON response content (if available)
```

Handling Exceptions

`requests` can raise exceptions like `ConnectionError`, `Timeout`, etc. It's good practice to handle these in your code.

```
try:
    response = requests.get(url, timeout=5)
    response.raise_for_status()  # Raises HTTPError for bad status codes
except requests.exceptions.HTTPError as errh:
    print(f'Http Error: {errh}')
except requests.exceptions.ConnectionError as errc:
    print(f'Error Connecting: {errc}')
except requests.exceptions.Timeout as errt:
    print(f'Timeout Error: {errt}')
except requests.exceptions.RequestException as err:
    print(f'OOps: Something Else: {err}')
```

Session Objects

For making several requests to the same host, the session object allows you to persist certain parameters across requests.

```python
with requests.Session() as session:
    session.headers.update({'x-test': 'true'})

    response = session.get(url)
    print(response.headers)

    response = session.get(url, headers={'x-test2': 'true'})
    print(response.headers)
```

To test HTTP requests, it's often useful to have access to a mock REST API server. Fortunately, there are several online resources available that provide this functionality. These services allow you to simulate various API responses without setting up your own backend. Here are some of the best online platforms for this purpose:

1. **HTTPBin** (http://httpbin.org)
    - HTTPBin offers a simple way to send requests to different endpoints and receive predictable responses. It supports various methods like GET, POST, PUT, DELETE, etc. It's very useful for testing HTTP libraries, such as `requests` in Python.
2. **JSONPlaceholder** (https://jsonplaceholder.typicode.com)
    - JSONPlaceholder is a free online REST API that you can use to fetch and manipulate fake data in the form of JSON. It's great for testing and prototyping your applications. It mimics a real RESTful API.
3. **Reqres** (https://reqres.in)
    - Reqres is a simple-to-use REST API that simulates real application scenarios. It provides endpoints for CRUD operations, making it ideal for testing front-end applications and learning how to interact with RESTful services.
4. **Mocky.io** (https://www.mocky.io)
    - Mocky allows you to design your own mock HTTP responses, specifying the status code, response body, and headers. It's useful for simulating responses from external servers and services.
5. **Beeceptor** (https://beeceptor.com)
    - Beeceptor provides an easy way to create custom endpoints for testing REST APIs. You can intercept HTTP requests and create mock rules for responses, making it very flexible for testing different scenarios.
6. **Postman Echo** (https://docs.postman-echo.com)
    - Postman Echo is a service that can be used to test REST clients and make sample API calls. It provides endpoints for various HTTP request methods and supports query parameters, request body and headers.
7. **FakeJSON** (https://fakejson.com)
    - FakeJSON is a service that provides fake data for testing and prototyping. While the basic service is free, it also offers more extensive features on a paid basis.
8. **MockServer** (https://www.mock-server.com)
    - MockServer allows you to mock any server or service via HTTP or HTTPS, including REST and SOAP services. It's more advanced and can be used both locally and as a remote service.

Notes

- **SSL Cert Verification**: By default, `requests` verifies SSL certificates for HTTPS requests. You can bypass this (not recommended for production) using `verify=False` in your request call.
- **Timeouts**: It's always a good idea to specify a timeout for your request, to avoid hanging indefinitely if the server does not respond.
- **Authentication**: The `requests` library supports various authentication mechanisms. You can pass an `auth` parameter (as a tuple of username and password) for basic HTTP authentication.
- **Custom Headers**: You can customize headers by passing a `headers` dictionary to your request call.
- **Stream Large Responses**: For large files, you can stream the download and process it in chunks with `stream=True` in your request.

`requests` is a powerful and intuitive library that simplifies HTTP requests in Python, making it easier to interact with web services and APIs.

Web Scraping using BeautifulSoup4

Web scraping is the process of extracting data from websites. Python, with its powerful libraries like Beautiful Soup, makes web scraping an accessible task. Beautiful Soup is a Python library for parsing HTML and XML documents, making it easy to scrape information from web pages. Here's a comprehensive guide with examples.

Installation

Before you start, you need to install Beautiful Soup and a parser library (like `lxml` or `html5lib`). Additionally, you'll often need `requests` to fetch web pages.

```
pip install beautifulsoup4 lxml requests
```

Basic Usage

1. **Import Libraries**

   First, import the necessary libraries.

   ```
   import requests
   from bs4 import BeautifulSoup
   ```

2. **Fetch a Web Page**

   Use `requests` to fetch the content of a web page.

   ```
   url = 'http://amazon.in'
   response = requests.get(url)
   ```

3. **Create a Soup Object**

   Parse the content of the page using Beautiful Soup.

```
soup = BeautifulSoup(response.content, 'lxml')
```

Examples of Web Scraping

1. **Extracting Titles**

   To get the title of the web page:

```
title = soup.title.text
print(title)
```

2. **Finding Elements**

   Beautiful Soup provides methods like `find()` and `find_all()` to locate elements.

   - To find the first instance of a `div`:

```
first_div = soup.find('div')
print(first_div)
```

   - To find all instances of a `p` tag:

```
paragraphs = soup.find_all('p')
for paragraph in paragraphs:
    print(paragraph.text)
```

3. **Extracting Links**

   To get all hyperlinks from a page:

```
links = soup.find_all('a')
for link in links:
    print(link.get('href'))
```

4. **Navigating the DOM Tree**

   You can navigate the DOM tree using tag names.

```
# Access the first `body` tag, then the first `div` inside it
body_div = soup.body.div
print(body_div)
```

5. **Using CSS Selectors**

   Beautiful Soup also allows you to use CSS selectors with the `select()` method.

```python
# Select all elements with the class 'some-class'
elements = soup.select('.some-class')
for element in elements:
    print(element.text)
```

6. **Handling Nested Tags**

   If you need to extract data from nested tags, you can drill down into the soup object.

```python
for div in soup.find_all('div', class_='container'):
    for p in div.find_all('p'):
        print(p.text)
```

Best Practices and Notes

- **Check Website's Terms and Conditions**: Before scraping a website, ensure that it's legal and ethical to do so. Check the site's terms and conditions or `robots.txt` file.
- **Handle Exceptions**: Websites may not always respond as expected. Handle network exceptions and HTTP errors.
- **Respect the Website**: To avoid overloading the server, space out your requests. Also, consider using caching to avoid re-fetching the same content.
- **Use a Browser User-Agent**: Some websites may block requests from Python's default user agent. You can set a user-agent to mimic a browser.

```python
headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)'}
response = requests.get(url, headers=headers)
```

- **Parsing JavaScript-Rendered Pages**: If the content you need is loaded by JavaScript, Beautiful Soup won't see it, because it only parses the static HTML content. In such cases, you might need tools like Selenium.
- **Regular Expressions**: For complex parsing, you might need to use regular expressions with Beautiful Soup.