Request Module Interview question

<https://www.geeksforgeeks.org/response-headers-python-requests/?ref=lbp>

To perform API testing in Python using the `requests` module, you can follow these steps. The `requests` library simplifies the process of sending HTTP requests and handling responses, making it straightforward to interact with APIs. Here’s a basic guide on how to get started:

### Step-by-Step Guide to API Testing with `requests`

#### 1. Install `requests`

If you haven't already installed the `requests` module, you can install it using pip:

```bash

pip install requests

### 2. Make GET Request

To send a GET request to an API endpoint and retrieve data:

import requests

# Example API endpoint

url = 'https://jsonplaceholder.typicode.com/posts/1'

# Send GET request

response = requests.get(url)

# Check if the request was successful (HTTP status code 200)

if response.status\_code == 200:

# Print response JSON data

print(response.json())

else:

print('Failed to retrieve data:', response.status\_code)

```

#### 3. Make POST Request

To send a POST request with data:

import requests

# Example API endpoint

url = 'https://jsonplaceholder.typicode.com/posts'

# Sample data to send with POST request

data = {

'title': 'foo',

'body': 'bar',

'userId': 1

}

# Send POST request

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

# Check if the request was successful (HTTP status code 201 for created)

if response.status\_code == 201:

# Print response JSON data

print(response.json())

else:

print('Failed to create data:', response.status\_code)

```

#### 4. Handling Authentication

If the API requires authentication (e.g., using API keys, tokens, or basic authentication), you can pass these credentials in the request headers:

```python

import requests

url = 'https://api.example.com/data'

headers = {

'Authorization': 'Bearer YOUR\_TOKEN\_HERE'

}

response = requests.get(url, headers=headers)

if response.status\_code == 200:

print(response.json())

else:

print('Failed to retrieve data:', response.status\_code)

```

#### 5. Handling Errors and Exceptions

Always handle potential errors and exceptions that may occur during API requests, such as network issues, server errors, or invalid responses:

import requests

url = 'https://api.example.com/data'

try:

response = requests.get(url)

response.raise\_for\_status() # Raise an HTTPError for bad responses (4xx or 5xx status codes)

if response.status\_code == 200:

print(response.json())

else:

print('Failed to retrieve data:', response.status\_code)

except requests.exceptions.RequestException as e:

print('Error during request:', e)

```

#### 6. Additional Methods (PUT, DELETE, etc.)

For other HTTP methods like PUT, DELETE, PATCH, etc., use `requests.put()`, `requests.delete()`, `requests.patch()` respectively, following a similar pattern as shown for GET and POST requests.

### Summary

The `requests` module in Python provides a simple yet powerful way to interact with APIs. By leveraging its functions and methods, you can easily perform various types of API testing, including sending requests, handling responses, and verifying data. Remember to handle errors gracefully and to structure your tests to cover different scenarios and edge cases as needed by your application or API specifications.

Certainly! Here are some interview questions related to using the `requests` module in Python for API testing or general HTTP request handling:

1. \*\*What is `requests` module in Python?\*\*

- Explain what the `requests` module is used for and its key features.

2. \*\*How do you send a GET request using `requests`?\*\*

- Describe the steps and necessary code to send a simple GET request to an API endpoint.

3. \*\*How do you handle authentication with `requests`?\*\*

- Explain how you would handle authentication (e.g., basic authentication, token-based authentication) when making requests using `requests`.

4. \*\*What HTTP methods does `requests` support, and how do you use them?\*\*

- List and briefly explain the HTTP methods (`GET`, `POST`, `PUT`, `DELETE`, etc.) supported by `requests`. Provide examples of how each method is used.

5. \*\*How do you handle response status codes in `requests`?\*\*

- Describe how you handle different HTTP status codes returned by API responses and why it's important to do so.

6. \*\*What are the main advantages of using `requests` over Python's built-in `urllib` module?\*\*

- Compare `requests` with `urllib` in terms of ease of use, features, and community support.

7. \*\*How do you handle exceptions and errors in `requests`?\*\*

- Explain the approach to handling exceptions and errors that can occur during HTTP requests with `requests`.

8. \*\*How would you send a POST request with JSON data using `requests`?\*\*

- Provide a code example demonstrating how to send a POST request with JSON data using `requests`.

9. \*\*What are HTTP headers, and how do you set custom headers with `requests`?\*\*

- Define HTTP headers and explain how to include custom headers in requests using `requests`.

10. \*\*How do you handle timeouts and retries with `requests`?\*\*

- Discuss strategies for setting timeouts and implementing retries when making HTTP requests with `requests`.

11. \*\*What is session object in `requests`? How is it useful?\*\*

- Explain what a session object is in `requests` and describe its benefits, especially in scenarios involving multiple requests to the same endpoint.

12. \*\*How do you perform file uploads with `requests`?\*\*

- Describe how you would upload files using `requests`, including handling multipart/form-data requests.

13. \*\*What is the purpose of `verify` parameter in `requests`? When would you use it?\*\*

- Explain the `verify` parameter in `requests`, its purpose, and when you might need to use it, especially in the context of HTTPS requests and SSL certificates.

14. \*\*How do you install and import `requests` module in Python?\*\*

- Outline the steps to install the `requests` module using `pip` and how to import it into a Python script or interactive session.

15. \*\*Can you describe a scenario where you would use `requests` in a real-world application?\*\*

- Provide an example of a practical use case or project where you would utilize `requests` for making API calls or handling HTTP requests.

These questions cover various aspects of using the `requests` module in Python, from basic usage to more advanced topics like authentication, error handling, and performance optimization. Preparing for these questions should give you a solid understanding of how to effectively utilize `requests` for API testing or any HTTP request handling needs in Python.

Certainly! Here's a structured answer to a common interview question about using the `requests` module in Python for API testing or HTTP request handling:

\*\*Question:\*\* How would you send a GET request to an API endpoint using the `requests` module in Python? Provide an example.

To send a GET request using the `requests` module in Python, you follow these steps:

1. \*\*Import the `requests` module:\*\* Start by importing the `requests` library, which simplifies sending HTTP requests and handling responses.

import requests

2. \*\*Specify the API endpoint:\*\* Define the URL of the API endpoint you want to send the GET request to.

url = 'https://jsonplaceholder.typicode.com/posts/1'

Replace `'https://jsonplaceholder.typicode.com/posts/1'` with the actual URL of the API endpoint you are working with.

3. \*\*Send the GET request:\*\* Use the `requests.get()` function to send the GET request to the specified URL.

response = requests.get(url)

This sends a GET request to the URL stored in the `url` variable and stores the response object in the `response` variable.

4. \*\*Process the response:\*\* Check the status code of the response to ensure the request was successful (`200` indicates success). Then, access the response content, which is typically in JSON format, using `response.json()`.

if response.status\_code == 200:

data = response.json()

print("Response JSON:", data)

else:

print(f"Failed to retrieve data. Status code: {response.status\_code}")

Here, `response.status\_code` checks if the request was successful. If successful (`status\_code == 200`), `response.json()` parses the JSON response content into a Python dictionary (`data`), which you can then manipulate or use as needed.

5. \*\*Complete example:\*\*

import requests

url = 'https://jsonplaceholder.typicode.com/posts/1'

try:

response = requests.get(url)

if response.status\_code == 200:

data = response.json()

print("Response JSON:", data)

else:

print(f"Failed to retrieve data. Status code: {response.status\_code}")

except requests.exceptions.RequestException as e:

print('Error during request:', e)

This example demonstrates how to send a GET request to an API endpoint using `requests`, handle the response, and manage potential errors or exceptions that may occur during the request process.

This answer provides a clear and structured approach to handling a common interview question about using the `requests` module in Python. It covers essential steps such as importing the module, sending a request, processing the response, and handling errors, showcasing proficiency in API interaction with `requests`.