Python Requests Library: A Beginner's Guide

Introduction

The Python Requests library is one of the most popular and user-friendly ways to interact with web services and APIs. It abstracts away the complexities of making HTTP requests, allowing you to focus on working with the data rather than managing the connection details.

This guide will walk you through the basics of using the Requests library, from installation to sending requests and processing responses.

Why Use Requests?

- **Simple API**: Clean, intuitive interface that's easy to learn
- **Built-in Features**: JSON parsing, custom headers, authentication, and more
- **Widely Adopted**: Industry standard for HTTP requests in Python
- **Great Documentation**: Extensive resources and community support

Installation

Before you can use Requests, you need to install it:

```
pip install requests
```

Making Your First Request

Let's start with a basic GET request, which retrieves data from a specified URL:

```python
import requests

# Make a simple GET request
response = requests.get('https://jsonplaceholder.typicode.com/todos/1')

# Print the response status code
print(f"Status code: {response.status_code}")

# Print the response content
print(f"Response content: {response.text}")
```

This code sends a GET request to a dummy API and prints both the status code (which tells you if the request was successful) and the response text.

Understanding the Response Object

The `response` object contains all the information returned by the server:

```python
import requests

response = requests.get('https://jsonplaceholder.typicode.com/users/1')

# Status code (200 means success)
print(f"Status code: {response.status_code}")

# Response headers (metadata about the response)
print(f"Content type: {response.headers['Content-Type']}")

# Raw content (bytes)
print(f"Raw content type: {type(response.content)}")
print(f"Raw content preview: {response.content[:50]}...")

# Text content (decoded string)
print(f"Text content type: {type(response.text)}")
print(f"Text content preview: {response.text[:50]}...")

# JSON content (parsed into Python objects)
json_data = response.json()
print(f"JSON data type: {type(json_data)}")
print(f"User name: {json_data['name']}")
```

Common HTTP Methods

Requests supports all standard HTTP methods:

GET Request (Retrieve Data)

```
# Basic GET request
response = requests.get('https://jsonplaceholder.typicode.com/posts/1')

# GET with query parameters
params = {'userId': 1, 'completed': 'true'}
response = requests.get('https://jsonplaceholder.typicode.com/todos', params=params)
```

POST Request (Create Data)

```
# Creating a new resource
new_todo = {
    'userId': 1,
    'title': 'Learn the Requests library',
    'completed': False
}

response = requests.post('https://jsonplaceholder.typicode.com/todos', json=new_todo)
print(f"Created todo ID: {response.json().get('id')}")
```

PUT Request (Update Data)

```
# Complete replacement of a resource
updated_todo = {
    'userId': 1,
    'id': 1,
    'title': 'Updated todo item',
    'completed': True
}

response = requests.put('https://jsonplaceholder.typicode.com/todos/1', json=updated_todo)
```

PATCH Request (Partial Update)

```
# Partial update of a resource
patch_data = {
    'title': 'Partially updated todo'
}

response = requests.patch('https://jsonplaceholder.typicode.com/todos/1', json=patch_data)
```

DELETE Request (Remove Data)

```
# Deleting a resource
response = requests.delete('https://jsonplaceholder.typicode.com/todos/1')
print(f"Delete status code: {response.status_code}")
```

Working with Headers

HTTP headers provide metadata about the request or response. You can both send and receive headers:

```
# Sending custom headers
headers = {
    'User-Agent': 'My Python App/1.0',
    'Accept': 'application/json'
}

response = requests.get('https://api.github.com/user', headers=headers)

# Reading response headers
for key, value in response.headers.items():
    print(f"{key}: {value}")
```

Handling Authentication

Many APIs require authentication:

```
# Basic authentication
response = requests.get('https://api.example.com/protected', auth=('username', 'password'))

# Or using a token (common for modern APIs)
headers = {'Authorization': 'Bearer your_token_here'}
response = requests.get('https://api.example.com/protected', headers=headers)
```

Error Handling

It's important to handle potential errors in your requests:

```
import requests
from requests.exceptions import HTTPError, ConnectionError, Timeout

try:
    response = requests.get('https://api.example.com/data', timeout=3)

    # Raise an exception for 4XX/5XX status codes
    response.raise_for_status()

    # Process the response...
    data = response.json()

except HTTPError as e:
    print(f"HTTP error occurred: {e}")
except ConnectionError as e:
    print(f"Network error occurred: {e}")
except Timeout as e:
    print(f"Request timed out: {e}")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

Working with JSON Data

Most modern APIs use JSON for data exchange:

```python
import requests
import json

# Sending JSON data
new_post = {
    'title': 'My New Post',
    'body': 'This is the content of my post.',
    'userId': 1
}

response = requests.post('https://jsonplaceholder.typicode.com/posts', json=new_post)

# Receiving and processing JSON data
post_data = response.json()
print(f"Created post with ID: {post_data['id']}")
print(f"Title: {post_data['title']}")

# Pretty-print JSON for debugging
print(json.dumps(post_data, indent=2))
```

Sessions

Sessions are useful for making multiple requests to the same site:

```python
import requests

# Create a session
session = requests.Session()

# This sets cookies, headers, etc. for all requests made with this session
session.headers.update({'User-Agent': 'My Python App/1.0'})

# First request sets cookies automatically
session.get('https://example.com/login')

# Subsequent requests will use the same cookies, headers, etc.
response = session.get('https://example.com/dashboard')
```

Timeouts

Always set timeouts to prevent your application from hanging indefinitely:

```
# Timeout in seconds
try:
    # Will raise a Timeout exception if the server doesn't respond in 5 seconds
    response = requests.get('https://api.example.com/slow-endpoint', timeout=5)
except requests.exceptions.Timeout:
    print("The request timed out")
```

Query Parameters

Many APIs use query parameters to filter or paginate results:

```
# Adding query parameters to a request
params = {
    'page': 2,
    'count': 10,
    'sort': 'name',
    'order': 'asc'
}

response = requests.get('https://api.example.com/users', params=params)

# The URL will be: https://api.example.com/users?page=2&count=10&sort=name&order=asc
print(f"Request URL: {response.url}")
```

Common Patterns and Best Practices

1. Check Status Code Before Processing

```
response = requests.get('https://api.example.com/data')

if response.status_code == 200:
    data = response.json()
    # Process data...
elif response.status_code == 404:
    print("Resource not found")
else:
    print(f"Error: Status code {response.status_code}")
```

2. Use raise_for_status()

```
response = requests.get('https://api.example.com/data')
response.raise_for_status()  # Raises HTTPError for 4XX/5XX status codes
# Process response...
```

3. Safe JSON Parsing

```
response = requests.get('https://api.example.com/data')

try:
    data = response.json()
except ValueError:
    print("Response is not valid JSON")
```

4. Retrying Failed Requests

For production systems, consider using a library like `requests-retry` to automatically retry failed requests with exponential backoff.

Debugging Tips

If you're having trouble with requests, try these debugging techniques:

```python
response = requests.get('https://api.example.com/data')

# Check the exact URL that was requested
print(f"Request URL: {response.url}")

# Check the status code
print(f"Status code: {response.status_code}")

# Look at the response headers
print(f"Headers: {dict(response.headers)}")

# Examine the raw response text
print(f"Response text: {response.text}")
```

Conclusion

The Requests library makes HTTP requests in Python simple and intuitive. By understanding the basics covered in this guide, you're well-prepared to interact with web services and APIs in your Python applications.

Remember that the official documentation at https://docs.python-requests.org/ is an excellent resource for more advanced usage and features.

Further Resources

- Requests Documentation
- HTTP Status Codes
- JSON Placeholder (Free fake API for testing)
- Postman (Tool for testing APIs outside of code)