
# Request module for HTTP request

**In the realm of web development, efficient communication between different servers and clients is paramount. The HTTP protocol stands as the foundation of this communication. Python, being a versatile language, offers several modules to facilitate HTTP requests, and among them, the "requests" module shines brightly.**

## Introduction to the Request Module

**The requests module is an elegant and intuitive HTTP library for Python, designed to simplify the process of making HTTP requests and handling responses. It abstracts the complexities of the underlying HTTP protocols into simple Pythonic functions and objects, making it a favorite among developers for its simplicity and versatility.**

## Installation

**Before diving into the wonders of the requests module, you need to ensure it's installed. You can easily install it via pip, the Python package installer:**

```
pip install requests
```

## Making GET Requests

**Let's start with the basics: making a GET request. The requests module provides a `get()` function, which allows you to retrieve data from a specified URL:**


```python
import requests

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

## Handling Response

**Once a request is made, the response object contains various attributes and methods to access the response data, headers, status code, etc. Here's a glimpse of how you can access some of this information:**

```python
import requests

response = requests.get('https://api.example.com/data')
print("Status Code:", response.status_code)
print("Response Body:", response.text)
print("Headers:", response.headers)
```

## Making POST Requests

**Besides GET requests, the requests module supports other HTTP methods like POST, PUT, DELETE, etc. For example, to make a POST request and send data, you can use the `post()` function:**

```python
import requests

data = {'key': 'value'}
response = requests.post('https://api.example.com/post', data=data)
print(response.text)
```


## Handling Errors

**In real-world scenarios, errors are inevitable. The requests module simplifies error handling by raising exceptions for common HTTP errors. You can catch these exceptions and handle them gracefully:**

```
import requests

try:
    response = requests.get('https://api.example.com/nonexistent')
    response.raise_for_status()
except requests.HTTPError as err:
    print("HTTP error occurred:", err)
except Exception as err:
    print("An unexpected error occurred:", err)
```

## Advanced Usage

**The requests module offers a plethora of advanced features like session management, authentication, cookies, proxies, SSL verification, and more. Here's a glimpse of session management:**

```
import requests

# Create a session
session = requests.Session()

# Perform multiple requests within the same session
response1 = session.get('https://api.example.com/resource1')
response2 = session.get('https://api.example.com/resource2')

# Close the session
session.close()
```


**CODE:**

```
import requests


# Making a GET request
def make_get_request(url):
    try:
        response = requests.get(url)
        # Check if the request was successful
(status code 200)
        if response.status_code == 200:
            print("GET Request Successful")
            print("Response Body:")
            print(response.text)
        else:
            print("GET Request Failed with
Status Code:", response.status_code)
    except requests.exceptions.RequestException
as e:
        print("An error occurred:", e)


# Making a POST request
def make_post_request(url, data):
    try:
        response = requests.post(url, data=data)
        # Check if the request was successful
(status code 200)
        if response.status_code == 200:
            print("POST Request Successful")
            print("Response Body:")
```


```
            print(response.text)
        else:
            print("POST Request Failed with
Status Code:", response.status_code)
    except requests.exceptions.RequestException
as e:
        print("An error occurred:", e)


# Example usage
if __name__ == "__main__":
    # Example URLs
    get_url =
'https://jsonplaceholder.typicode.com/posts/1'
    post_url =
'https://jsonplaceholder.typicode.com/posts'

    # Making a GET request
    print("Making a GET Request...")
    make_get_request(get_url)
    print("\n")

    # Making a POST request
    print("Making a POST Request...")
    post_data = {'title': 'foo', 'body': 'bar',
'userId': 1}
    make_post_request(post_url, data=post_data)
```


## Output:

```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\41.py
Making a GET Request...
GET Request Successful
Response Body:
{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum es
}


Making a POST Request...
POST Request Failed with Status Code: 201
```

## JSON MODULE:

**In the vast landscape of data handling and exchange, JSON (JavaScript Object Notation) stands tall as a lightweight and human-readable data interchange format. Python, being a versatile language, offers a built-in module called `json` to seamlessly work with JSON data. Let's delve into the world of Python's JSON module to understand how it simplifies the process of data serialization and deserialization.**

## Introduction to the JSON Module

**The `json` module in Python provides functions to serialize (convert Python objects into JSON strings) and deserialize (convert JSON strings into Python objects) data. It acts as a bridge between Python's data structures and JSON.**

## Serialization: Python to JSON

**Serialization involves converting Python objects into JSON strings. The `json.dumps()` function accomplishes this task effortlessly. It**


**accepts a Python object as input and returns its JSON representation as a string.**

```
import json

# Python dictionary
data = {'name': 'John', 'age': 30, 'city': 'New York'}

# Serialize Python dictionary to JSON
json_data = json.dumps(data)
print(json_data)
```

## Deserialization: JSON to Python

**Deserialization is the reverse process, where JSON strings are converted back into Python objects. The `json.loads()` function achieves this transformation, taking a JSON string as input and returning the corresponding Python object.**

```
import json

# JSON string
json_data = '{"name": "John", "age": 30, "city": "New York"}'

# Deserialize JSON to Python dictionary
data = json.loads(json_data)
print(data)
```

## Handling Complex Data Types

**Python's JSON module seamlessly handles complex data types such as lists, tuples, dictionaries, integers, floats, strings, and even custom**

**objects. It automatically converts them to their JSON equivalents and vice versa.**

```
import json

# Python list of dictionaries
people = [{'name': 'John', 'age': 30}, {'name': 'Jane', 'age': 25}]

# Serialize Python list to JSON
json_data = json.dumps(people)
print(json_data)
```

## Pretty Printing

**The JSON module also provides a `json.dumps()` parameter, `indent`, which enables pretty printing of JSON data, making it more human-readable.**

```
import json

# Python dictionary
data = {'name': 'John', 'age': 30, 'city': 'New York'}

# Pretty print JSON data
json_data = json.dumps(data, indent=4)
print(json_data)
```

## Error Handling

**The `json` module provides error handling mechanisms to deal with invalid JSON data during deserialization. It raises `ValueError` exceptions when encountering invalid JSON syntax.**


```python
import json

# Invalid JSON string
invalid_json = '{"name": "John", "age": 30, "city": "New York"'

try:
    # Attempting to deserialize invalid JSON
    data = json.loads(invalid_json)
except ValueError as e:
    print("Error:", e)
```

**CODE:**

```python
import json

# Sample Python dictionary
data = {
    "name": "John",
    "age": 30,
    "city": "New York",
    "is_student": False,
    "courses": ["Math", "Science", "History"],
    "address": {
        "street": "123 Main St",
        "city": "Anytown",
        "zipcode": "12345"
    }
}

# Serialize Python dictionary to JSON string
json_data = json.dumps(data, indent=4)
print("Serialized JSON data:")
```


```
print(json_data)

# Deserialize JSON string to Python dictionary
decoded_data = json.loads(json_data)
print("\nDeserialized Python dictionary:")
print(decoded_data)
```

**Output:**

```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\41_1.py
Serialized JSON data:
{
    "name": "John",
    "age": 30,
    "city": "New York",
    "is_student": false,
    "courses": [
        "Math",
        "Science",
        "History"
    ],
    "address": {
        "street": "123 Main St",
        "city": "Anytown",
        "zipcode": "12345"
    }
}

Deserialized Python dictionary:
{'name': 'John', 'age': 30, 'city': 'New York', 'is_student': False, 'courses': ['Math', 'Science', 'History'], 'address': {'street': '12
```