Web APIs, Python Requests & Performing an HTTP Request in Python Tutorial

Learn about the basics of HTTP and also about the request library in Python to make different types of requests.

Updated Feb 2023 · 15 min read

CONTENTS

SHARE

***Practice performing an HTTP request in Python* with this hands-on exercise.**

Application Programming Interfaces (APIs) are software mediators; their job is to permit applications to communicate with each other. These subtle mediators appear in everyday life whether you know it or not. For example, if you've sent an instant message today, you've used an API.

More specifically, APIs allow people to send and retrieve data using code. However, It's more common to use APIs to retrieve data; for instance, you can read this blog post because your web browser retrieved the data that makes up this page from the DataCamp server.

But web servers don't randomly send data - that would be like going to a restaurant, and the waiter randomly brings you a meal. A request must be made to the server to retrieve data before it responds with data. This is true for the waiter in a restaurant, and if you'd like to retrieve some data from an API – you make an API request to a server, and it will respond with the appropriate data.

The de facto industry standard for sending HTTP requests in Python is the requests library. There is also Python's built-in urllib, but Pythonistas tend to prefer the python requests API due to its readability and the fact it supports fully restful APIs – something we will touch on a little later.

The requests library isolates all of the challenges of making requests behind a straightforward API - this allows you to concentrate on communicating with services and consuming data in your application.

In this article, we will walk through some of the core components of the requests library and provide some code examples to help you get started.

REST APIs

We've established APIs are software mediators. Another way to think of them is as a type of software interface that grants other applications access to specific data and methods.

One of the most popular architectures used to build APIs is the REpresentational State Transfer (REST) pattern. The REST architectural design enables the client and server to be implemented independently of one another without being aware of each other - this means code on either side can be changed without worrying about how the change will affect the other.

They are a set of guidelines designed to simplify communications between software, thereby making the process of accessing data more straightforward and logical. Don't worry if you don't know these guidelines; you don't need to know them to get started – what you do need to know is how data is exposed from REST services.

Data from REST web services are exposed to the internet through a public URL, which can be accessed by sending an HTTP request.

HTTP Methods

Let's rewind to our restaurant analogy; to order food at a restaurant, the waiter will approach you, and you say what you want. The waiter then passes your request to the chef, who makes the meal and passes it to the waiter to return to you. In other words, the chef wouldn't cook your meal until your request has been sent.

REST APIs are the same: they listen for HTTP request methods before taking any action. HTTP is what defines a set of request methods to tell the API what operations to perform for a given resource. It specifies how to interact with the resources located at the provided endpoint.

There are several HTTP methods, but five are commonly used with REST APIs:

HTTP Method	Description
GET	Retrieve data
POST	Create data
PUT	Update existing data
PATCH	Partially update existing data
DELETE	Delete data

It's highly likely you will be performing GET requests more than any other method in data

analysis and data science. This is down to the fact that it's the most necessary method required to gain access to certain datasets – learn how to do this with DataCamp's **Intermediate Importing Data in Python course**.

When you perform a request to a web server, a response is returned by the API. Attached to the response is an HTTP status code. The purpose of the status code is to provide additional information about the response, so the client knows the type of request being received.

Note: *Learn more about **Status Codes**.*

Endpoints

The data you interact with on a web server is delineated with a URL. Much like how a web page URL is connected to a single page, an endpoint URL is connected to particular resources within an API. Therefore, an endpoint may be described as a digital location where an API receives inquiries about a particular resource on its server – think of it as the other end of a communication channel.

To add more context, REST APIs expose a set of public URLs that may be requested by client applications to access the resources of the web service. The public URLs exposed by the REST API are known as "endpoints."

Using Python to Consume APIs

The Python requests API enables developers to write code to interact with REST APIs. It allows them to send HTTP requests using Python without having to worry about the complexities that typically come with carrying out such tasks (i.e., manually adding query strings to URLs, form-encoding `PUT` and `POST` data, etc.).

Despite being considered the de facto standard for making HTTP requests in Python, the `requests` module is not part of Python's standard library – it must be installed.

The most straightforward way to install the requests module is with **pip**:

```
python -m pip install requests
```

Explain code — Powered by OpenAI

It's always recommended to manage the Python packages required for different projects in **virtual environments**; this way, the packages for one project will not interfere and break system tools in other projects because they are isolated – instead of being installed globally.

Now we've got the requests module installed, let's see how it works.

Making a GET request

We've already established GET is one of the most common HTTP request methods you'll encounter when working with REST APIs. It allows you (the client) to retrieve data from web servers.

An important thing to note is GET is a read-only operation meaning it's only suitable for accessing existing resources but should not be used to modify them.

To demonstrate how the request module works, we will use **JSONPlaceholder**, which is a freely available fake API used for testing and prototyping.

Follow along with the code in this **DataCamp Workspace**.

```
import requests

# The API endpoint
url = "https://jsonplaceholder.typicode.com/posts/1"

# A GET request to the API
response = requests.get(url)

# Print the response
response_json = response.json()
print(response_json)

"""
{'userId': 1, 'id': 1, 'title': 'sunt aut facere repellat provident occaecati exc
"""
```

Explain code — Powered by OpenAI

In the code above, we carried out the following:

1. Defined the API endpoint to retrieve data from
2. Used the `requests.get(url)` method to retrieve the data from the defined endpoint.
3. We used the `response.json()` method to store the response data in a dictionary object; note that this only works because the result is written in JSON format – an error would have been raised otherwise.
4. The last step is to print the JSON response data.

We can also check the status code returned from the API like this:

```
# Print status code from original response (not JSON)
print(response.status_code)

"""
200
"""
```

Explain code — Powered by OpenAI

You can also pass arguments to a python GET request. To do this, we must slightly alter the code above. Here's how the new code looks...

```
# The API endpoint
url = "https://jsonplaceholder.typicode.com/posts/"

# Adding a payload
payload = {"id": [1, 2, 3], "userId":1}

# A get request to the API
response = requests.get(url, params=payload)

# Print the response
response_json = response.json()

for i in response_json:
    print(i, "\n")

"""
{'userId': 1, 'id': 1, 'title': 'sunt aut facere repellat provident occaecati exc

{'userId': 1, 'id': 2, 'title': 'qui est esse', 'body': 'est rerum tempore vitae\

{'userId': 1, 'id': 3, 'title': 'ea molestias quasi exercitationem repellat qui i
"""
```

Explain code — Powered by OpenAI

Here's what we did differently:

1. Changed the API endpoint. Notice it no longer has a '1' at the end.
2. Defined the payload in a dictionary.
3. Passed the payload to the `param` argument of the `requests.get()` method.
4. This returned a list object so we looped through the list and printed each item on a new line.

Making a POST request

GET requests allow you to retrieve data; POST requests allow you to create new data. Let's take a look at how we can create new data on the JSONPlaceholder server.

```
# Define new data to create
new_data = {
    "userID": 1,
    "id": 1,
    "title": "Making a POST request",
    "body": "This is the data we created."
}

# The API endpoint to communicate with
url_post = "https://jsonplaceholder.typicode.com/posts"

# A POST request to tthe API
post_response = requests.post(url_post, json=new_data)

# Print the response
post_response_json = post_response.json()
print(post_response_json)

"""
{'userID': 1, 'id': 101, 'title': 'Making a POST request', 'body': 'This is the d
"""
```

Explain code — Powered by OpenAI

In the code above, we performed the following:

1. Created a new resource we wanted to add to the JSONPlaceholder API
2. Defined the endpoint to POST the new data
3. Sent a POST request using the `requests.post()` method. Note that the `json` parameter was set in the `post()` method; we do this to tell the API we are explicitly sending a JSON object to the specified URL.
4. Used the `response.json()` method to store the response data in a dictionary object
5. The last step is to print the JSON response data.

WAIT!

Before you read the next bit of code, take 20 seconds to think about what status code we can expect to be returned by the API.

Remember, this time, we created a new resource instead of simply retrieving it.

Okay, here it goes...

```
# Print status code from original response (not JSON)
print(post_response.status_code)

"""
201
"""
```

Explain code — Powered by OpenAI

Did you get it right?

Advanced topics

Authenticating requests

Up to this point, the interactions we've had with the REST API have been pretty straightforward. The JSONPlaceholder API does not require any authentication for you to start interacting with it. But, there are several instances where a REST API may require authentication before access is granted to specific endpoints – especially when you're dealing with sensitive data.

For example, if you want to create integrations, retrieve data, and automate your workflows on GitHub, you can do so with GitHub REST API. However, there are many operations on the GitHub REST API that require authentication, such as retrieving public and private information about authenticated users.

Here's a simple workaround using the Python requests module:

```
from requests.auth import HTTPBasicAuth

private_url = "https://api.github.com/user"
github_username = "username"
token = "token"

private_url_response = requests.get(
    url=private_url,
    auth=HTTPBasicAuth(github_username, token)
)

private_url_response.status_code

"""
200
"""
```

Explain code — Powered by OpenAI

In the code above we:

1. Imported the `HTTPBasicAuth` object from `requests.auth`; this object attaches HTTP basic authentication to the given request object – it's essentially the same as typing your username and password into a website.
2. Defined the private URL endpoint to access
3. Instantiated a variable with a GitHub username – we anonymized the username for privacy.
4. Instantiated a variable GitHub with a personal access token for authentication.
5. Retrieved data from our endpoint and stored it in the `private_url_response` variable.
6. Displayed the status code.

Handling errors

There are instances where requests made to an API do not go as expected. Several factors on either the client or server-side could be at play. Regardless of the cause, the outcome is always the same: the request fails.

When using REST APIs, it's always a good idea to make your code resilient. However, before

you can write robust code, you must understand how to manage the reported errors when things do not go to plan.

Let's go back to the JSONPlaceholder API for this demonstration. We will start by writing some code and then explain what is going on.

```
# A deliberate typo is made in the endpoint "postz" instead of "posts"
url = "https://jsonplaceholder.typicode.com/postz"

# Attempt to GET data from provided endpoint
try:
    response = requests.get(url)
    response.raise_for_status()
# If the request fails (404) then print the error.
except requests.exceptions.HTTPError as error:
  print(error)

"""
404 Client Error: Not Found for url: https://jsonplaceholder.typicode.com/postz
"""
```

Explain code — Powered by OpenAI

In the code above, we:

1. Defined the JSONPlace holder endpoint to retrieve data from, but we made a deliberate typo when constructing the URL – this will raise a 404 error.
2. Used Python's built-in **exception handling**, try and except catch any errors that occur when attempting to visit the JSONPlaceholder endpoint. Note, the `raise_for_status()` method is what is used to return an HTTPError object when an error occurs during the process.
3. Printed the error that was raised.

Although we demonstrated how to handle 404 error status codes in this instance, the same format can be used to handle any HTTP status code.

Dealing with too many redirects

HTTP status codes with the 3xx format indicate the client was redirected and must perform some additional actions to complete the request. However, this can occasionally lead to situations where you end up with an infinite redirect loop.

Python's requests module provides the TooManyRedirects object to handle this problem, as follows:

```
"""
Note: The code here will not raise an error
but the structure is how you would hand a case where there
are multiple redirects
"""

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

try:
  response = requests.get(url)
  response.raise_for_status()
except requests.exceptions.TooManyRedirects as error:
  print(error)
```

Explain code — Powered by OpenAI

You can also set the maximum number of redirects as a parameter of your HTTP request method:

```
# Solution 2
url = "https://jsonplaceholder.typicode.com/posts"
session = requests.Session()
session.max_redirects = 3
response = session.get(url)
```

Explain code — Powered by OpenAI

Another option is to completely disable redirects:

```
# Solution 3
url = "https://jsonplaceholder.typicode.com/posts"
session = requests.Session()
session.allow_redirects = False
response = session.get(url)
```

Explain code — Powered by OpenAI

Connection errors

These are another sort of error you may face when attempting to send requests to a server. There are several reasons you may not receive a response from the server (i.e., DNS failure, refused connection, internet connection issues, etc.), but the outcome is consistent: a connection error is raised.

You can use the `requests` modules ConnectionError exception object to catch these issues and handle them accordingly.

Here's how the code would look:

```
"""
Note: The code here will not raise an error
but the structure is how you would hand a case where there
is a connection error.
"""

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

try:
  response = requests.get(url)
except requests.ConnectionError as error:
  print(error)
```

Explain code — Powered by OpenAI

Timeout

When the API server accepts your connection but cannot finish your request in the allowed time, you will get what is known as a "timeout error."

We will demonstrate how to handle this case by setting the `timeout` parameter in the `requests.get()` method to an extremely small number; this will raise an error and we will handle that error using the `requests.Timeout` object.

```
url = "https://jsonplaceholder.typicode.com/posts"

try:
  response = requests.get(url, timeout=0.0001)
except requests.Timeout as error:
  print(error)
```

Explain code — Powered by OpenAI

The most straightforward workaround for timeout errors is to set longer timeouts. Other

Kurtis Pykes
Data Science & AI Blogger | Top 1000 Medium Writers on AI and Data Science

TOPICS

Python

Top 26 Python pandas Interview Questions and Answers

Navigating TensorFlow Certification: The TensorFlow Developer Certificate

solutions may include optimizing your requests, incorporating a retry loop into your scripts, or performing asynchronous API calls – a technique that allows your software to begin a potentially long-running activity while being responsive to other events rather than having to wait until that task is completed.

Wrap up

In this tutorial, we covered what APIs are and explored a common API architecture called REST. We also looked at HTTP methods and how we can use the Python requests library to interact with web services.

Check out the following courses to develop your data science skills:

- Streamlined Data Ingestion with Pandas
- Analyzing Social Media Data in Python
- Intermediate Importing Data in Python

TOPICS

Python

Anaconda vs Python: Exploring Their Key Differences

Mastering Shiny for Python: A Beginner's Guide to Building Interactive Web Applications

Python Courses

COURSE

Introduction to Python

4 hr 4.8M

Master the basics of data analysis with Python in just four hours. This online course will introduce the Python interface and explore popular packages.

See Details → Start Course

COURSE

Introduction to Data Science in Python

4 hr 433.6K

Dive into data science using Python and learn how to effectively analyze and visualize your data. No coding experience or skills needed.

See Details → Start Course

COURSE

Intermediate Python

4 hr 927.9K

Level up your data science skills by creating visualizations using Matplotlib and manipulating DataFrames with pandas.

See Details → Start Course

See More →

Related

Top 26 Python pandas Interview Questions and Answers

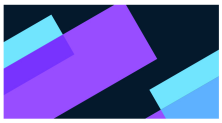
Explore key Python pandas interview questions and answers for data science roles

Srujana Maddula • 15 min

Navigating TensorFlow Certification: The TensorFlow...

Explore TensorFlow certification with this guide. Learn about its importance, preparation, and how DataCamp's course...

Matt Crabtree • 8 min



Anaconda vs Python: Exploring Their Key Differences

Learn all about the key differences between Python and Anaconda in this complete guide.

Austin Chia • 9 min

Mastering Shiny for Python: A Beginner's Guide to Building...

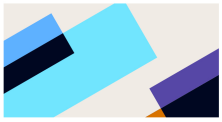
Explore the basics of Shiny for Python so you can start making interactive dashboards and web applications with th...

Amberle McKee

How to Make a Seaborn Histogram: A Detailed Guide

Find out how to create a histogram chart using the Seaborn library in Python.

Austin Chia • 9 min



MongoDB Tutorial: How to Set Up and Query MongoDB...

Learn how to connect and analyze MongoDB databases in Python.

Bex Tuychiev • 10 min

See More →