

# Python's Requests Library (Guide)

by [Alex Ronquillo](#) Feb 28, 2024

[intermediate](#) [web-dev](#)

Mark as Completed

Share

## Table of Contents

- [Getting Started With Python's Requests Library](#)
- [The GET Request](#)
- [The Response](#)
  - [Status Codes](#)
  - [Content](#)
  - [Headers](#)
- [Query String Parameters](#)
- [Request Headers](#)
- [Other HTTP Methods](#)
- [The Message Body](#)
- [Request Inspection](#)
- [Authentication](#)
- [SSL Certificate Verification](#)
- [Performance](#)
  - [Timeouts](#)
  - [The Session Object](#)
  - [Max Retries](#)
- [Conclusion](#)

Deploy your python data apps for free

Try **Posit Connect Cloud** Now

[Remove ads](#)

Watch Now

This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Making HTTP Requests With Python](#)

The [Requests](#) library is the de facto standard for making HTTP requests in Python. It abstracts the complexities of making requests behind a beautiful, simple API so that

— FREE Email Series —

[Python Tricks](#)

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

Email...

Get Python Tricks »

No spam. Unsubscribe any time.

Browse Topics

Guided Learning Paths

Basics

Intermediate

Advanced

api

best-practices

career

community

databases

data-science

data-structures

data-viz

devops

django

docker

editors

flask

front-end

gamedev

gui

machine-learning

numpy

projects

python

testing

tools

web-dev

web-scraping

Deploy your python data apps for free

Posit Connect CLOUD

posit

## Table of Contents

- [Getting Started With Python's Requests Library](#)
- [The GET Request](#)
- [The Response](#)
- [Query String Parameters](#)
- [Request Headers](#)
- [Other HTTP Methods](#)
- [The Message Body](#)

you can focus on interacting with services and consuming data in your application.

Throughout this tutorial, you'll see some of the most useful features that Requests has to offer as well as ways to customize and optimize those features for different situations that you may come across. You'll also learn how to use Requests in an efficient way as well as how to prevent requests to external services from slowing down your application.

### In this tutorial, you'll learn how to:

- **Make requests** using the most common HTTP methods
- **Customize** your requests' headers and data using the query string and message body
- **Inspect** data from your requests and responses
- Make **authenticated** requests
- **Configure** your requests to help prevent your application from backing up or slowing down

For the best experience working through this tutorial, you should have [basic general knowledge of HTTP](#). That said, you still may be able to follow along fine without it.

In the upcoming sections, you'll see how you can install and use `requests` in your application. If you want to play with the code examples that you'll see in this tutorial, as well as some additional ones, then you can download the code examples and work with them locally:

**Get Your Code:** [Click here to download the free sample code](#) that shows you how to use Python's Requests library.

- [Request Inspection](#)
  - [Authentication](#)
  - [SSL Certificate Verification](#)
  - [Performance](#)
- [Conclusion](#)

Mark as Completed




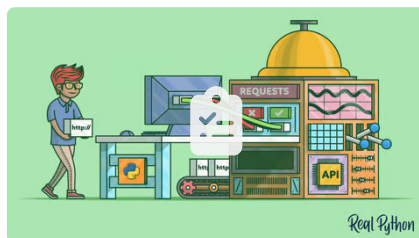
Share

#### Recommended Video Course

[Making HTTP Requests With Python](#)



 **Take the Quiz:** Test your knowledge with our interactive “HTTP Requests With the "requests" Library” quiz. You'll receive a score upon completion to help you track your learning progress:



#### Interactive Quiz

#### [HTTP Requests With the "requests" Library](#)

Test your understanding of the Python "requests" library for making HTTP requests and interacting with web services.

## Getting Started With Python's Requests Library

Even though the Requests library is a common staple for many Python developers, it's not included in [Python's standard library](#). There are [good reasons for that decision](#), primarily that the library can continue to evolve more freely as a self-standing project.

**Note:** Requests doesn't support asynchronous HTTP requests directly. If you need [async](#) support in your program, you should try out [AIOHTTP](#) or [HTTPX](#). The latter library is broadly compatible with Requests' syntax.

Because Requests is a third-party library, you need to install it before you can use it in your code. As a good practice, you should install external packages into a [virtual environment](#), but you may choose to install `requests` into your global environment if you're planning to use it across multiple projects.

Whether you're working in a virtual environment or not, you'll need to install

requests:

Shell



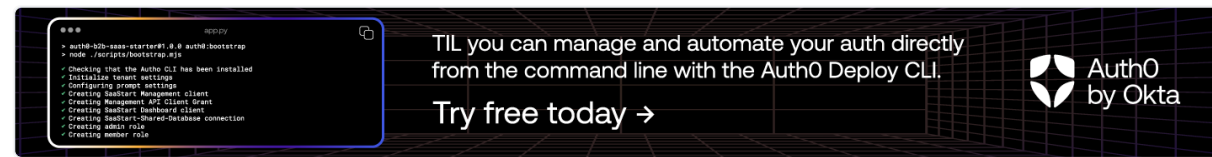
```
$ python -m pip install requests
```

Once [pip](#) has finished installing requests, you can use it in your application. Importing requests looks like this:

Python

```
import requests
```

Now that you're all set up, it's time to begin your journey through Requests. Your first goal will be learning how to make a GET request.



[Remove ads](#)

## The GET Request

[HTTP methods](#), such as GET and POST, determine which action you're trying to perform when making an HTTP request. Besides GET and POST, there are several other common methods that you'll use later in this tutorial.

One of the most common HTTP methods is GET. The GET method indicates that you're trying to get or retrieve data from a specified resource. To make a GET request using Requests, you can invoke `requests.get()`.

To test this out, you can make a GET request to [GitHub's REST API](#) by calling `get()` with the following URL:

Python



```
>>> import requests
>>> requests.get("https://api.github.com")
<Response [200]>
```

Congratulations! You've made your first request. Now you'll dive a little deeper into the response of that request.

## The Response

A Response is a powerful object for inspecting the results of the request. Make that same request again, but this time store the return value in a [variable](#) so that you can get a closer look at its attributes and behaviors:

Python



```
>>> import requests
>>> response = requests.get("https://api.github.com")
```

In this example, you've captured the return value of `get()`, which is an instance of Response, and stored it in a variable called `response`. You can now use `response` to see a lot of information about the results of your GET request.

## Status Codes

The first bit of information that you can gather from Response is the status code. A status code informs you of the status of the request.

For example, a 200 OK status means that your request was successful, whereas a 404 NOT FOUND status means that the resource you were looking for wasn't found. There are [many other possible status codes](#) as well to give you specific insights into what happened with your request.

By accessing `.status_code`, you can see the status code that the server returned:

Python



```
>>> response.status_code
200
```

`.status_code` returned 200, which means that your request was successful and the server responded with the data that you were requesting.

Sometimes, you might want to use this information to make decisions in your code:

Python

```
if response.status_code == 200:
    print("Success!")
elif response.status_code == 404:
    print("Not Found.")
```

With this logic, if the server returns a 200 status code, then your program will [print](#) Success!. If the result is a 404, then your program will print Not Found.

Requests goes one step further in simplifying this process for you. If you use a Response instance in a conditional expression, then it'll evaluate to True if the status code was smaller than 400, and False otherwise.

Therefore, you can simplify the last example by rewriting the if statement:

Python

```
if response:
    print("Success!")
else:
    raise Exception(f"Non-success status code: {response.status_code}")
```

In the code snippet above, you implicitly check whether the `.status_code` of response is between 200 and 399. If it's not, then you [raise](#) an [exception](#) that includes the non-success status code in an [f-string](#).

**Note:** This [truth value test](#) is possible because `__bool__()` is an [overloaded method](#) on Response. This means that the adapted default behavior of Response takes the status code into account when determining the truth value of the object.

Keep in mind that this method is *not* verifying that the status code is equal to 200. The reason for this is that other status codes within the 200 to 399 range, such as 204 NO CONTENT and 304 NOT MODIFIED, are also considered successful in the sense that they provide some workable response.

For example, the status code 204 tells you that the response was successful, but there's no content to return in the message body.

So, make sure you use this convenient shorthand only if you want to know if the request was generally successful. Then, if necessary, you'll need to handle the response appropriately based on the status code.

Let's say you don't want to check the response's status code in an if statement. Instead, you want to use Request's built-in capacities to raise an exception if the

request was unsuccessful. You can do this using `.raise_for_status()`:

Python

raise\_error.py


```
import requests
from requests.exceptions import HTTPError

URLS = ["https://api.github.com", "https://api.github.com/invalid"]


for url in URLS:
    try:
        response = requests.get(url)
        response.raise_for_status()
    except HTTPError as http_err:
        print(f"HTTP error occurred: {http_err}")
    except Exception as err:
        print(f"Other error occurred: {err}")
    else:
        print("Success!")
```


If you invoke `.raise_for_status()`, then Requests will raise an `HTTPError` for status codes between 400 and 600. If the status code indicates a successful request, then the program will proceed without raising that exception.

Now, you know a lot about how to deal with the status code of the response that you got back from the server. However, when you make a GET request, you rarely only care about the status code of the response. Usually, you want to see more. Next, you'll see how to view the actual data that the server sent back in the body of the response.



TIL you can do all of this for free with Auth0  
 PLUS you get 5 Organizations for your B2B app.  
 Try free today →



 [Remove ads](#)

## Content

The response of a GET request often has some valuable information, known as a [payload](#), in the message body. Using the attributes and methods of `Response`, you can view the payload in a variety of different formats.

To see the response's content in [bytes](#), you use `.content`:

Python



```
>>> import requests

>>> response = requests.get("https://api.github.com")
>>> response.content
b'{"current_user_url":"https://api.github.com/user", ...}'

>>> type(response.content)
<class 'bytes'>
```

While `.content` gives you access to the raw bytes of the response payload, you'll often want to convert them into a [string](#) using a [character encoding](#) such as [UTF-8](#). `response.text` will do that for you when you access `.text`:

Python



```
>>> response.text
'{"current_user_url":"https://api.github.com/user", ...}'

>>> type(response.text)
<class 'str'>
```



Because the decoding of bytes to a `str` requires an encoding scheme, Requests will try to guess the [encoding](#) based on the response's [headers](#) if you don't specify one. You can provide an explicit encoding by setting `.encoding` before accessing `.text`:

Python



```
>>> response.encoding = "utf-8" # Optional: Requests infers this.
>>> response.text
'{"current_user_url": "https://api.github.com/user", ...}'
```

If you take a look at the response, then you'll see that it's actually serialized JSON content. To get a dictionary, you could take the `str` that you retrieved from `.text` and deserialize it using [`json.loads\(\)`](#). However, a simpler way to accomplish this task is to use `.json()`:

Python



```
>>> response.json()
{'current_user_url': 'https://api.github.com/user', ...}

>>> type(response.json())
<class 'dict'>
```

The type of the return value of `.json()` is a dictionary, so you can access values in the object by key:

Python



```
>>> response_dict = response.json()
>>> response_dict["emojis_url"]
'https://api.github.com/emojis'
```

You can do a lot with status codes and message bodies. But, if you need more information, like [metadata](#) about the response itself, then you'll need to look at the response's headers.

## Headers

The response headers can give you useful information, such as the content type of the response payload and a time limit on how long to cache the response. To view these headers, access `.headers`:

Python



```
>>> import requests

>>> response = requests.get("https://api.github.com")
>>> response.headers
{'Server': 'GitHub.com',
 ...
 'X-GitHub-Request-Id': 'AE83:3F40:2151C46:438A840:65C38178'}
```

`.headers` returns a dictionary-like object, allowing you to access header values by key. For example, to see the content type of the response payload, you can access `"Content-Type"`:

Python



```
>>> response.headers["Content-Type"]
'application/json; charset=utf-8'
```

There's something special about this dictionary-like headers object, though. The HTTP specification defines headers as case-insensitive, which means that you're able to access these headers without worrying about their capitalization:

Python



```
>>> response.headers["content-type"]  
'application/json; charset=utf-8'
```

Whether you use the key "content-type" or "Content-Type", you'll get the same value.

Now that you've seen the most useful attributes and methods of Response in action, you already have a good overview of Requests' basic usage. You can get content from the Internet and work with the response that you receive.

But there's more to the Internet than plain and straightforward URLs. In the next section, you'll take a step back and see how your responses change when you customize your GET requests to account for query string parameters.



Master **Real-World Python Skills**  
With a **Community of Experts**  
Level Up With Unlimited Access to Our Vast Library  
of Python Tutorials and Video Lessons

[Watch Now »](#)

 [Remove ads](#)

## Query String Parameters

One common way to customize a GET request is to pass values through [query string](#) parameters in the URL. To do this using `get()`, you pass data to `params`. For example, you can use GitHub's [repository search](#) API to look for popular Python repositories:

Python

search\_popular\_repos.py

```
import requests  
  
# Search GitHub's repositories for popular Python projects  
response = requests.get(  
    "https://api.github.com/search/repositories",  
    params={"q": "language:python", "sort": "stars", "order": "desc"},  
)  
  
# Inspect some attributes of the first three repositories  
json_response = response.json()  
popular_repositories = json_response["items"]  
for repo in popular_repositories[:3]:  
    print(f"Name: {repo['name']}")  
    print(f"Description: {repo['description']}")  
    print(f"Stars: {repo['stargazers_count']}")  
    print()
```

By passing a dictionary to the `params` parameter of `get()`, you're able to modify the results that come back from the search API.

You can pass `params` to `get()` in the form of a dictionary, as you've just done, or as a list of tuples:

Python



```
>>> import requests  
  
>>> requests.get(  
...     "https://api.github.com/search/repositories",  
...     [("q", "language:python"), ("sort", "stars"), ("order", "desc")],  
... )  
<Response [200]>
```

You can even pass the values as bytes:

Python



```
>>> requests.get(
...     "https://api.github.com/search/repositories",
...     params=b"q=language:python&sort=stars&order=desc",
... )
<Response [200]>
```

Query strings are useful for parameterizing GET requests. Another way to customize your requests is by adding or modifying the headers that you send.

## Request Headers

To customize headers, you pass a dictionary of HTTP headers to `get()` using the `headers` parameter. For example, you can change your previous search request to highlight matching search terms in the results by specifying the `text-match` media type in the `Accept` header:

Python

text\_matches.py

```
import requests

response = requests.get(
    "https://api.github.com/search/repositories",
    params={"q": "real python"},
    headers={"Accept": "application/vnd.github.text-match+json"},
)

# View the new `text-matches` list which provides information
# about your search term within the results
json_response = response.json()
first_repository = json_response["items"][0]
print(first_repository["text_matches"][0]["matches"])
```

The `Accept` header tells the server what content types your application can handle. In this case, since you're expecting the matching search terms to be highlighted, you're using the header value `application/vnd.github.text-match+json`, which is a proprietary GitHub `Accept` header where the content is a special JSON format.

If you run this code, then you'll get a result similar to the one shown below:

Shell



```
$ python text_matches.py
[{'text': 'Real Python', 'indices': [23, 34]}]
```

Before you learn more ways to customize requests, you'll broaden your horizons by exploring other HTTP methods.

## Other HTTP Methods

Aside from GET, other popular HTTP methods include POST, PUT, DELETE, HEAD, PATCH, and OPTIONS. For each of these HTTP methods, Requests provides a function, with a similar signature to `get()`.

**Note:** To try out these HTTP methods, you'll make requests to [httpbin.org](https://httpbin.org). The `httpbin` service is a great resource created by the original author of Requests, [Kenneth Reitz](https://kennethreitz.com). The service accepts test requests and responds with data about the requests.

You'll notice that Requests provides an intuitive interface to all the mentioned HTTP methods:



Python



```
>>> import requests

>>> requests.get("https://httpbin.org/get")
<Response [200]>
>>> requests.post("https://httpbin.org/post", data={"key": "value"})
<Response [200]>
>>> requests.put("https://httpbin.org/put", data={"key": "value"})
<Response [200]>
>>> requests.delete("https://httpbin.org/delete")
<Response [200]>
>>> requests.head("https://httpbin.org/get")
<Response [200]>
>>> requests.patch("https://httpbin.org/patch", data={"key": "value"})
<Response [200]>
>>> requests.options("https://httpbin.org/get")
<Response [200]>
```

In the example code above, you called each function to make a request to the httpbin service using the corresponding HTTP method.

**Note:** All of these functions are high-level shortcuts to `requests.request()`, passing the name of the relevant HTTP method:

Python



```
>>> requests.request("GET", "https://httpbin.org/get")
<Response [200]>
```

You could use the equivalent lower-level function call, but the power of Python's Requests library lies in its human-friendly high-level interface.

You can inspect the responses in the same way as you did before:

Python



```
>>> response = requests.head("https://httpbin.org/get")
>>> response.headers["Content-Type"]
'application/json'

>>> response = requests.delete("https://httpbin.org/delete")
>>> json_response = response.json()
>>> json_response["args"]
{}
```

Headers, response bodies, status codes, and more are returned in the Response for each method.

Next you'll take a closer look at the POST, PUT, and PATCH methods and learn how they differ from the other request types.

 [The Real Python Podcast »](#) [Remove ads](#)

## The Message Body

According to the HTTP specification, POST, PUT, and the less common PATCH requests pass their data through the message body rather than through parameters in the query string. Using Requests, you'll pass the payload to the corresponding function's data parameter.

data takes a dictionary, a list of tuples, bytes, or a file-like object. You'll want to adapt the data that send in the body of your request to the specific needs of the service that you're interacting with.

For example, if your request's content type is `application/x-www-form-urlencoded`, then you can send the form data as a dictionary:

Python



```
>>> import requests

>>> requests.post("https://httpbin.org/post", data={"key": "value"})
<Response [200]>
```

You can also send that same data as a list of tuples:

Python



```
>>> requests.post("https://httpbin.org/post", data=[("key", "value")])
<Response [200]>
```

If, however, you need to send JSON data, then you can use the `json` parameter. When you pass JSON data via `json`, Requests will serialize your data and add the correct Content-Type header for you.

Like you learned earlier, the `httpbin` service accepts test requests and responds with data about the requests. For instance, you can use it to inspect a basic POST request:

Python



```
>>> response = requests.post("https://httpbin.org/post", json={"key": "value"})
>>> json_response = response.json()
>>> json_response["data"]
'{"key": "value"}'
>>> json_response["headers"]["Content-Type"]
'application/json'
```

You can see from the response that the server received your request data and headers as you sent them. Requests also provides this information to you in the form of a `PreparedRequest` that you'll inspect in more detail in the next section.

## Request Inspection

When you make a request, the Requests library prepares the request before actually sending it to the destination server. Request preparation includes things like validating headers and serializing JSON content.

You can view the `PreparedRequest` object by accessing `.request` on a `Response` object:

Python



```
>>> import requests

>>> response = requests.post("https://httpbin.org/post", json={"key": "value"})

>>> response.request.headers["Content-Type"]
'application/json'
>>> response.request.url
'https://httpbin.org/post'
>>> response.request.body
b'{"key": "value"}'
```

Inspecting `PreparedRequest` gives you access to all kinds of information about the request being made, such as payload, URL, headers, authentication, and more.

So far, you've made a lot of different kinds of requests, but they've all had one thing in common: they're unauthenticated requests to public APIs. Many services you may come across will want you to authenticate in some way.

## Authentication

Authentication helps a service understand who you are. Typically, you provide your credentials to a server by passing data through the `Authorization` header or a custom header defined by the service. All the functions of Requests that you've seen to this point provide a parameter called `auth`, which allows you to pass your credentials:

Python



```
>>> import requests

>>> response = requests.get(
...     "https://httpbin.org/basic-auth/user/passwd",
...     auth=("user", "passwd")
... )

>>> response.status_code
200
>>> response.request.headers["Authorization"]
'Basic dXNlcjpwYXNzd2Q='
```

The request succeeds if the credentials that you pass in the tuple to `auth` are valid.

When you pass your credentials in a tuple to the `auth` parameter, Requests applies the credentials using HTTP's [Basic access authentication scheme](#) under the hood.

The Basic Authentication Scheme

Show/Hide

You could make the same request by passing explicit Basic authentication credentials using `HTTPBasicAuth`:

Python



```
>>> from requests.auth import HTTPBasicAuth
>>> requests.get(
...     "https://httpbin.org/basic-auth/user/passwd",
...     auth=HTTPBasicAuth("user", "passwd")
... )
<Response [200]>
```

Though you don't need to be explicit for Basic authentication, you may want to authenticate using another method. Requests provides [other methods of authentication](#) out of the box, such as `HTTPDigestAuth` and `HTTPProxyAuth`.

A real-world example of an API that requires authentication is GitHub's [authenticated user](#) API. This endpoint provides information about the authenticated user's profile.

If you try to make a request without credentials, then you'll see that the status code is `401 Unauthorized`:

Python



```
>>> requests.get("https://api.github.com/user")
<Response [401]>
```

If you don't provide authentication credentials when accessing a service that requires them, then you'll get an HTTP error code as a response.

To make a request to GitHub's authenticated user API, you first need to [generate a personal access token](#) with the [read:user scope](#). Then you can pass this token as the second element in a tuple to `get()`:

Python



```
>>> import requests

>>> token = "<YOUR_GITHUB_PA_TOKEN>"
>>> response = requests.get(
...     "https://api.github.com/user",
...     auth=("", token)
... )
>>> response.status_code
200
```

Like you learned previously, this approach passes the credentials to `HTTPBasicAuth`, which expects a username and a password and sends the credentials as a Base64-encoded string with the prefix `"Basic "`:

Python



```
>>> response.request.headers["Authorization"]
'Basic 0mdocF92dkd...WpremM0SGRuUGY='
```

This works, but it's not the right way to [authenticate with a Bearer token](#)—and using an empty string input for the superfluous username is awkward.

With Requests, you can supply your own authentication mechanism to fix that. To try this out, create a subclass of `AuthBase` and implement `__call__()`:

Python

custom\_token\_auth.py

```
from requests.auth import AuthBase

class TokenAuth(AuthBase):
    """Implements a token authentication scheme."""

    def __init__(self, token):
        self.token = token

    def __call__(self, request):
        """Attach an API token to the Authorization header."""
        request.headers["Authorization"] = f"Bearer {self.token}"
        return request
```

Here, your custom `TokenAuth` mechanism receives a token, then includes that token in the `Authorization` header of your request, also setting the recommended `"Bearer "` prefix to the string.

You can now use this custom token authentication to make your call to GitHub's authenticated user API:

Python



```
>>> import requests
>>> from custom_token_auth import TokenAuth

>>> token = "<YOUR_GITHUB_PA_TOKEN>"
>>> response = requests.get(
...     "https://api.github.com/user",
...     auth=TokenAuth(token)
... )

>>> response.status_code
200
>>> response.request.headers["Authorization"]
'Bearer ghp_b...Tx'
```

Your custom `TokenAuth` created a well-formatted string for the `Authorization` header. You can now use this more intuitive way of interacting with a token-based authentication scheme such as the one that parts of GitHub's API require.

**Note:** While you could construct the authentication string outside of a custom authentication class and pass it directly with `headers`, this approach is [discouraged](#) because it can lead to [unexpected behavior](#).

When you attempt to set your authentication credentials directly using `headers`, then Requests may internally overwrite your input. This can happen, for example, if you have a [.netrc file](#) that provides authentication credentials. Requests will attempt to [get the credentials from the .netrc file](#) if you don't provide an authentication method using `auth=`.

Bad authentication mechanisms can lead to security vulnerabilities. Unless a service requires a custom authentication mechanism for some reason, you'll always want to use a tried-and-true auth scheme like the built-in Basic authentication or [OAuth](#), for example through [Requests-OAuthlib](#).

While you're thinking about security, consider dealing with SSL certificates using Requests.



[Become a Python Expert »](#)

 [Remove ads](#)

## SSL Certificate Verification

Anytime the data that you're trying to send or receive is sensitive, security is important. The way that you communicate with secure sites over HTTP is by establishing an encrypted connection using SSL, which means that verifying the target server's SSL certificate is critical.

The good news is that Requests does this for you by default. However, there are some cases where you might want to change this behavior.

If you want to disable SSL certificate verification, then you pass `False` to the `verify` parameter of the request function:

Python





```
>>> import requests

>>> requests.get("https://api.github.com", verify=False)
InsecureRequestWarning: Unverified HTTPS request is being made to host
'api.github.com'. Adding certificate verification is strongly advised
See: https://urllib3.readthedocs.io/en/latest/advanced-usage.html#tls
warnings.warn(
<Response [200]>
```

Requests even warns you when you're making an insecure request to help you keep your data safe!

**Note:** [Requests uses a package called `certifi`](#) to provide certificate authorities. This lets Requests know which authorities it can trust. Therefore, you should update `certifi` frequently to keep your connections as secure as possible.

Now that you know how to make all sorts of HTTP requests using Requests, authenticated or not, you may wonder about how you can make sure that your program works as quickly as possible.

In the next section, you'll learn about a few ways that you can improve performance with the help of Requests.

## Performance

When using Requests, especially in a production application environment, it's important to consider performance implications. Features like timeout control, sessions, and retry limits can help you keep your application running smoothly.

### Timeouts

When you make an inline request to an external service, your system will need to wait for the response before moving on. If your application waits too long for that response, requests to your service could back up, your user experience could suffer, or your background jobs could hang.

By default, Requests will wait indefinitely on the response, so you should almost always specify a timeout duration to prevent these issues from happening. To set the request's timeout, use the `timeout` parameter. `timeout` can be an integer or float representing the number of seconds to wait on a response before timing out:

```
Python 

>>> requests.get("https://api.github.com", timeout=1)
<Response [200]>
>>> requests.get("https://api.github.com", timeout=3.05)
<Response [200]>
```

In the first request, the request will time out after 1 second. In the second request, the request will time out after 3.05 seconds.

[You can also pass a tuple](#) to `timeout` with the following two elements:

1. **Connect timeout:** The time it allows for the client to establish a connection to the server
2. **Read timeout:** The time it'll wait on a response once your client has established a connection

Both of these elements should be numbers, and can be of type `int` or `float`:

Python



```
>>> requests.get("https://api.github.com", timeout=(3.05, 5))
<Response [200]>
```

If the request establishes a connection within 3.05 seconds and receives data within 5 seconds of the connection being established, then the response will be returned as it was before. If the request times out, then the function will raise a `Timeout` exception:

Python

timeout\_catcher.py

```
import requests
from requests.exceptions import Timeout

try:
    response = requests.get("https://api.github.com", timeout=(3.05, 5))
except Timeout:
    print("The request timed out")
else:
    print("The request did not time out")
```

Your program can [catch the Timeout exception](#) and respond accordingly.

[Learn Python »](#) [Remove ads](#)

## The Session Object

Until now, you've been dealing with high-level `requests` APIs such as `get()` and `post()`. These functions are abstractions of what's going on when you make your requests. They hide implementation details, such as how connections are managed, so that you don't have to worry about them.

Underneath those abstractions is a class called `Session`. If you need to fine-tune your control over how requests are being made or improve the performance of your requests, you may need to use a `Session` instance directly.

Sessions are used to persist parameters across requests. For example, if you want to use the same authentication across multiple requests, then you can use a session:

Python

persist\_info\_with\_session.py

```
1 import requests
2 from custom_token_auth import TokenAuth
3
4 TOKEN = "<YOUR_GITHUB_PA_TOKEN>"
5
6 with requests.Session() as session:
7     session.auth = TokenAuth(TOKEN)
8
9     first_response = session.get("https://api.github.com/user")
10    second_response = session.get("https://api.github.com/user")
11
12 print(first_response.headers)
13 print(second_response.json())
```

In this code example, you use a [context manager](#) to ensure that the session releases the resources when it doesn't need them anymore.

In line 7, you log in using your custom `TokenAuth`. You only need to log in once per session, and then you can make multiple authenticated requests. Requests will persist the credentials while the session exists.

You then make two requests to the authenticated user API in lines 9 and 10 using `session.get()` instead of `get()`.

The primary performance optimization of sessions comes in the form of persistent connections. When your app makes a connection to a server using a `Session`, it keeps that connection around in a connection pool. When your app wants to connect to the same server again, it'll reuse a connection from the pool rather than establishing a new one.

## Max Retries

When a request fails, you may want your application to retry the same request. However, Requests won't do this for you by default. To apply this functionality, you need to implement a custom [transport adapter](#).

Transport adapters let you define a set of configurations for each service that you're interacting with. For example, say you want all requests to `https://api.github.com` to retry two times before finally raising a `RetryError`. You'd build a transport adapter, set its `max_retries` parameter, and mount it to an existing `Session`:

```
Python                                retry_twice.py

import requests
from requests.adapters import HTTPAdapter
from requests.exceptions import RetryError

github_adapter = HTTPAdapter(max_retries=2)

session = requests.Session()

session.mount("https://api.github.com", github_adapter)

try:
    response = session.get("https://api.github.com/")
except RetryError as err:
    print(f"Error: {err}")
finally:
    session.close()
```

In this example, you've set up your session so that it'll retry a maximum of two times when your request to GitHub's API doesn't work as expected.

When you mount the `HTTPAdapter`—in this case, `github_adapter`—to `session`, then `session` will adhere to its configuration for each request to `https://api.github.com`.

**Note:** While the implementation shown above works, you won't see any effect of the retry behavior unless there's something wrong with your network connection or GitHub's servers.

If you want to play around with code that builds on top of this example, and you'd like to inspect when the retries happen, then you're in luck. You can download the materials of this tutorial and take a look at `retry_thrice.py`:

**Get Your Code:** [Click here to download the free sample code](#) that shows you how to use Python's Requests library.

The code in this file improves on the example shown above by using the underlying [urllib3.util.Retry](#) to further customize the retry functionality. It also adds [logging](#) to display debugging output, which gives you a chance to monitor when


Python attempted the retries.

Requests comes packaged with intuitive implementations for timeouts, transport adapters, and sessions that can help you keep your code efficient and your application resilient.

### Find Your Dream Python Job

pythonjobshq.com



 [Remove ads](#)


## Conclusion

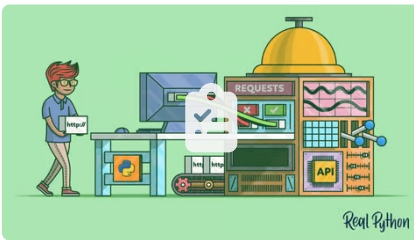
Nice work, you've made it to the end of the tutorial, and you've come a long way in increasing your knowledge about Python's powerful Requests library.

**In this tutorial, you've learned how to:**

- **Make requests** using a variety of different HTTP methods such as GET, POST, and PUT
- **Customize your requests** by modifying headers, authentication, query strings, and message bodies
- **Inspect the data** you send to the server and the data the server sends back to you
- Work with **SSL certificate verification**
- Use Requests effectively with `max_retries`, `timeout`, `sessions`, and transport adapters

Because you learned how to use Requests, you're equipped to explore the wide world of web services and build awesome applications using the fascinating data they provide.

 **Take the Quiz:** Test your knowledge with our interactive “HTTP Requests With the "requests" Library” quiz. You'll receive a score upon completion to help you track your learning progress:




#### Interactive Quiz

#### [HTTP Requests With the "requests" Library](#)

Test your understanding of the Python "requests" library for making HTTP requests and interacting with web services.



Mark as Completed



 Share

**Watch Now**

This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Making HTTP Requests With Python](#)

 Python Tricks 

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Send Me Python Tricks »

## About **Alex Ronquillo**



Alex Ronquillo is a Software Engineer at thelab. He's an avid Pythonista who is also passionate about writing and game development.

» [More about Alex](#)

*Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:*



[Aldren](#)



[Brad](#)



[Geir Arne](#)



[Joanna](#)



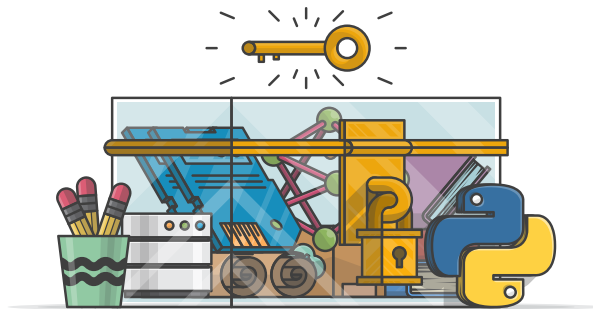
[Kate](#)



[Martin](#)

Master Real-World Python Skills  
With Unlimited Access to Real Python





**Join us and get access to thousands of tutorials, hands-on video courses, and a community of expert Pythonistas:**

Level Up Your Python Skills »

## What Do You Think?

Rate this article:



LinkedIn

Twitter

Bluesky

Facebook

Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

**Commenting Tips:** The most useful comments are those written with the goal of learning from or helping out other students. [Get tips for asking good questions](#) and [get answers to common questions in our support portal](#).

Looking for a real-time conversation? Visit the [Real Python Community Chat](#) or join the next [“Office Hours” Live Q&A Session](#). Happy Pythoning!

## Keep Learning

Related Topics: **intermediate** **web-dev**

Recommended Video Course: [Making HTTP Requests With Python](#)


Related Tutorials:

- [Beautiful Soup: Build a Web Scraper With Python](#)
- [Python and REST APIs: Interacting With Web Services](#)
- [Working With JSON Data in Python](#)
- [Object-Oriented Programming \(OOP\) in Python](#)
- [Speed Up Your Python Program With Concurrency](#)

## A Peer-to-Peer Learning Community for Python Enthusiasts...Just Like You

pythonistacafe.com



 [Remove ads](#)

© 2012–2025 Real Python · [Newsletter](#) · [Podcast](#) · [YouTube](#) · [Twitter](#) · [Facebook](#) · [Instagram](#) ·

[Python Tutorials](#) · [Search](#) · [Privacy Policy](#) · [Energy Policy](#) · [Advertise](#) · [Contact](#)

♥ Happy Pythoning!