

Real Python

Python's Requests Library (Guide)

by Alex Ronquillo ⌂ Jan 23, 2019 💬 16 Comments 📁 intermediate web-dev

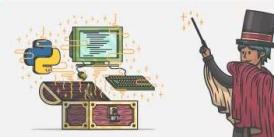
[Tweet](#) [Share](#) [Email](#)

Table of Contents

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

Improve Your Python with Python Tricks

realpython.com



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

Throughout this article, you'll see some of the most useful features that `requests` has to offer as well as how to customize and optimize those features for different situations 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 methods
- **Customize** your requests' headers
- **Inspect** data from your requests as JSON
- Make **authenticated** requests
- **Configure** your requests to help prevent errors

```
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}
```

Though I've tried to include as much information as possible in this article, I do assume a [very basic general knowledge](#) of Python anyway.

Now that that is out of the way, let's dive in and see how you can use `requests` in your application!

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

[Send Python Tricks »](#)

 Remove ads

Getting Started With `requests`

Let's begin by installing the `requests` library. To do so, run the following command:

Shell

```
$ pip install requests
```

If you prefer to use [Pipenv](#) for managing Python packages, you can run the following:

Shell

```
$ pipenv install requests
```

Once `requests` is installed, 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.

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, invoke `requests.get()`.

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

[Improve Your Python](#)

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

Congratulations! You've made your first request. Let's 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. Let's 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

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

In this example, you've captured the returned Response object and stored it in a variable called `response`. You can now use response to inspect the results of the request.

```
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}
```

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

Email Address

[Send Python Tricks »](#)

Status Codes

The first bit of information that you can access is the status code 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 was not 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 a `200`, which means your request was successful and the server responded with the data 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, your program will print `Success!`. If the result is a `404`, 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, it will evaluate to `True` if the status code was between `200` and `400`, and `False` otherwise.

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

Python

```
if response:
    print('Success!')
else:
    print('An error has occurred.')
```

[Improve Your Python](#)

Technical Detail: This [Truth Value Test](#) is made possible because `__bool__()` is an overloaded method on `Response`.

This means that the default behavior of `Response` has been redefined to take 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 400 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 204 tells you that the request was successful.

So, make sure you use this convenient snippet to check the status code, then, if necessary, handle the response.

Let's say you don't want to check the response status code, but instead exception if the request was unsuccessful.

```
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}
```

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

Python

```
import requests
from requests.exceptions import HTTPError

for url in ['https://api.github.com', 'https://api.github.com/invalid']:
    try:
        response = requests.get(url)

        # If the response was successful, no Exception will be raised
        response.raise_for_status()
    except HTTPError as http_err:
        print(f'HTTP error occurred: {http_err}') # Python 3.6
    except Exception as err:
        print(f'Other error occurred: {err}') # Python 3.6
    else:
        print('Success!')
```

[Send Python Tricks »](#)

If you invoke `.raise_for_status()`, an `HTTPError` will be raised for certain status codes. If the status code indicates a successful request, the program will proceed without that exception being raised.

Further Reading: If you're not familiar with Python 3.6's [f-strings](#), I encourage you to take advantage of them as they are a great way to simplify your formatted strings.

Now, you know a lot about how to deal with the status code of the response 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.

[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

```
>>> response = requests.get('https://api.github.com')
>>> response.content
b'{"current_user_url": "https://api.github.com/user", "current_user_authorizations_html_url": "https://github.com/{query}&page,per_page,sort,order", "commit_search_url": "https://api.github.com/search/commits?q={query}&page,per_page,sort,order", "emails_url": "https://api.github.com/user/emails", "emojis_url": "https://api.github.com/{query}&page,per_page,sort,order", "issues_url": "https://api.github.com/issues", "keys_url": "https://api.github.com/type,page,per_page,sort,order", "organization_url": "https://api.github.com/{query}&page,per_page,sort,order", "current_user_repos_url": "https://api.github.com/repositories"}'
```

```
type,page,per_page,sort}" , "starred_url": "https://api.github.com/user/starred{/owner}{/repo}" , "starred_gists_url": "https://api.github.com/gists/starred" , "team_url": "https://api.github.com/teams?type,page,per_page,sort}" , "user_search_url": "https://api.github.com/search/users?q={query}{&page,per_page,sort,order}" }'
```

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

Python

>>>

```
>>> response.text
'{"current_user_url": "https://api.github.com/user", "current_user_authorizations_html_url": "https://github.com/q={query}{&page,per_page,sort,order}" , "commit_search_url": "https://api.github.com/search/commits?q={query}{&page,per_page,sort,order}" , "emails_url": "https://api.github.com/user/emails" , "emojis_url": "https://api.github.com/q={query}{&page,per_page,sort,order}" , "issues_url": "https://api.github.com/issues" , "keys_url": "https://api.github.com/type,page,per_page,sort}" , "organization_url": "https://api.github.com/orgs/{org}" , "public_gists_url": "https://q={query}{&page,per_page,sort,order}" , "current_user_repositories_url": "https://api.github.com/user/repos?type,page,per_page,sort}" , "starred_url": "https://api.github.com/user/starred{/owner}{/repo}" , "starred_gists_url": "https://api.github.com/gists/starred" , "team_url": "https://api.github.com/teams?type,page,per_page,sort}" , "user_search_url": "https://api.github.com/search/users?q={query}{&page,per_page,sort,order}" }'
```

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

[Send Python Tricks »](#)

Because the decoding of bytes to a `str` is handled automatically by `response.json()`, you can skip specifying the response's `headers` if you do not specify one. You can provide an explicit encoding by setting `.encoding` before accessing `.text`:

Python

>>>

```
>>> response.encoding = 'utf-8' # Optional: requests infers this internally
>>> response.text
'{"current_user_url": "https://api.github.com/user", "current_user_authorizations_html_url": "https://github.com/q={query}{&page,per_page,sort,order}" , "commit_search_url": "https://api.github.com/search/commits?q={query}{&page,per_page,sort,order}" , "emails_url": "https://api.github.com/user/emails" , "emojis_url": "https://api.github.com/q={query}{&page,per_page,sort,order}" , "issues_url": "https://api.github.com/issues" , "keys_url": "https://api.github.com/type,page,per_page,sort}" , "organization_url": "https://api.github.com/orgs/{org}" , "public_gists_url": "https://q={query}{&page,per_page,sort,order}" , "current_user_repositories_url": "https://api.github.com/user/repos?type,page,per_page,sort}" , "starred_url": "https://api.github.com/user/starred{/owner}{/repo}" , "starred_gists_url": "https://api.github.com/gists/starred" , "team_url": "https://api.github.com/teams?type,page,per_page,sort}" , "user_search_url": "https://api.github.com/search/users?q={query}{&page,per_page,sort,order}" }'
```

If you take a look at the response, you'll see that it is actually serialized JSON content. To get a dictionary, you could take the `str` 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', 'current_user_authorizations_html_url': 'https://github.com/settings/connections/applications{/client_id}', 'authorizations_url': 'https://api.github.com/authorizations', 'code_search_url': 'https://api.github.com/search/code?q={query}{&page,per_page,sort,order}' , 'commit_search_url': 'https://api.github.com/search/commits?q={query}{&page,per_page,sort,order}' , 'emails_url': 'https://api.github.com/user/emails' , 'emojis_url': 'https://api.github.com/emojis' , 'events_url': 'https://api.github.com/events' , 'feeds_url': 'https://api.github.com/feeds' , 'followers_url': 'https://api.github.com/user/followers' , 'following_url': 'https://api.github.com/user/following{/target}' , 'gists_url': 'https://api.github.com/gists{/gist_id}' , 'hub_url': 'https://api.github.com/hub' , 'issue_search_url': 'https://api.github.com/search/issues?q={query}{&page,per_page,sort,order}' , 'issues_url': 'https://api.github.com/issues' , 'keys_url': 'https://api.github.com/user/keys' , 'notifications_url': 'https://api.github.com/notifications' }
```

```
organization_url': 'https://api.github.com/orgs/{org}', 'public_gists_url':
'https://api.github.com/gists/public', 'rate_limit_url': 'https://api.github.com/rate_limit',
'repository_url': 'https://api.github.com/repos/{owner}/{repo}', 'repository_search_url':
'https://api.github.com/search/repositories?q={query}&page,per_page,sort,order}',
'current_user_repositories_url': 'https://api.github.com/user/repos?type,page,per_page,sort',
'starred_url': 'https://api.github.com/user/starred{/owner}{/repo}', 'starred_gists_url':
'https://api.github.com/gists/starred', 'team_url': 'https://api.github.com/teams', 'user_url':
'https://api.github.com/users/{user}', 'user_organizations_url': 'https://api.github.com/user/orgs',
'user_repositories_url': 'https://api.github.com/users/{user}/repos?type,page,per_page,sort',
'user_search_url': 'https://api.github.com/search/users?q={query}&page,per_page,sort,order'}
```

The type of the return value of .json() is

You can do a lot with status codes and the response itself, you'll need to look at the

Headers

The response headers can give you useful limit on how long to cache the response

Python

```
>>> response.headers
{'Server': 'GitHub.com', 'Date': 'Mon, 10 Dec 2018 17:49:54 GMT', 'Content-Type': 'application/json; charset=utf-8', 'Transfer-Encoding': 'chunked', 'Status': '200 OK', 'X-RateLimit-Limit': '60', 'X-RateLimit-Remaining': '59', 'X-RateLimit-Reset': '1544467794', 'Cache-Control': 'public, max-age=60, s-maxage=60', 'Vary': 'Accept', 'ETag': 'W/"7dc470913f1fe9bb6c7355b50a0737bc"', 'X-GitHub-Media-Type': 'github.v3; format=json', 'Access-Control-Expose-Headers': 'ETag, Link, Location, Retry-After, X-GitHub-OTP, X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset, X-OAuth-Scopes, X-Accepted-OAuth-Scopes, X-Poll-Interval, X-GitHub-Media-Type', 'Access-Control-Allow-Origin': '*', 'Strict-Transport-Security': 'max-age=31536000; includeSubdomains; preload', 'X-Frame-Options': 'deny', 'X-Content-Type-Options': 'nosniff', 'X-XSS-Protection': '1; mode=block', 'Referrer-Policy': 'origin-when-cross-origin, strict-origin-when-cross-origin', 'Content-Security-Policy': "default-src 'none'", 'Content-Encoding': 'gzip', 'X-GitHub-Request-Id': 'E439:4581:CF2351:1CA3E06:5C0EA741'}
```

.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 is something special about this dictionary-like headers object, though. The HTTP spec defines headers to be case-insensitive, which means we are 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, you've learned the basics about Response. You've seen its most useful attributes and methods in action. Let's take a step back and see how your responses change when you customize your GET requests.

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

Email Address

[Send Python Tricks »](#)

[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 [Search API](#) to look for the `requests` library:

Python

```
import requests

# Search GitHub's repositories for `requests` + `language:python`
response = requests.get(
    'https://api.github.com/search/repositories',
    params={'q': 'requests+language:python'}
)

# Inspect some attributes of the `repository` object
json_response = response.json()
repository = json_response['items'][0]
print(f'Repository name: {repository["name"]}')
print(f'Repository description: {repository["description"]}'')
```

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

[Send Python Tricks »](#)

By passing the dictionary `{'q': 'requests+language:python'}` to the `params` parameter of `.get()`, you are 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 have just done, or as a list of tuples:

Python

```
>>> requests.get(
...     'https://api.github.com/search/repositories',
...     params=[('q', 'requests+language:python')],
... )
<Response [200]>
```

You can even pass the values as bytes:

Python

```
>>> requests.get(
...     'https://api.github.com/search/repositories',
...     params=b'q=requests+language:python',
... )
<Response [200]>
```

Query strings are useful for parameterizing GET requests. You can also customize your requests by adding or modifying the headers 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

```
import requests

response = requests.get(
    'https://api.github.com/search/repositories',
    params={'q': 'requests+language:python'},
    headers={'Accept': 'application/vnd.github.v3.text-match+json'},
)

# View the new `text-matches` array which provides information
# about your search term within the results
json_response = response.json()
```

[Improve Your Python](#)

```
repository = json_response[items][0] Python's Requests Library (Guide) - Real Python
print(f'Text matches: {repository["text_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.v3.text-match+json`, which is a proprietary GitHub Accept header where the content is a special JSON format.

Before you learn more ways to customize requests, let's broaden the horizon by exploring other HTTP methods.

Other HTTP Methods

Aside from GET, other popular HTTP method, with a similar signature to get(

Python

```
>>> requests.post('https://httpbin.org/post')
>>> requests.put('https://httpbin.org/put')
>>> requests.delete('https://httpbin.org/delete')
>>> requests.head('https://httpbin.org/head')
>>> requests.patch('https://httpbin.org/patch')
>>> requests.options('https://httpbin.org/options')
```

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

Email Address

[Send Python Tricks »](#)

Each function call makes a request to the httpbin service using the corresponding HTTP method. For each method, you can inspect their responses in the same way 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.

[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 you send in the body of your request to the specific needs of the service you're interacting with.

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

Python

```
>>> 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, 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.

httpbin.org is a great resource created by the author of `requests`, [Kenneth Reitz](#). It's a service that 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'

# How to merge two dicts
# 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}
```

You can see from the response that the `Content-Type` header provides this information to you in the form of JSON.

>>>

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

[Send Python Tricks »](#)

Inspecting Your Requests

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` by accessing `.request`:

Python

```
>>> 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 the `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 request functions you've seen to this point provide a parameter called `auth`, which allows you to pass your credentials.

One example of an API that requires authentication is GitHub's [Authenticated User](#) API. This endpoint provides information about the authenticated user's profile. To make a request to the Authenticated User API, you can pass your GitHub username and password in a tuple to `get()`:

Python

```
>>> from getpass import getpass
>>> requests.get('https://api.github.com/user', auth=('username', getpass()))
<Response [200]>
```

>>>

The request succeeded if the credentials you passed in the tuple to `auth` are valid. If you try to make this request with no credentials, you'll see that the status code is 401 Unauthorized.

[Improve Your Python](#)

Python

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

When you pass your username and password in a tuple to the auth parameter, requests is applying the credentials using HTTP's [Basic access authentication scheme](#) under the hood.

Therefore, you could make the same request by passing explicit Basic authentication credentials using `HTTPBasicAuth`:

Python

```
>>> from requests.auth import HTTPBasicAuth
>>> from getpass import getpass
>>> requests.get(
...     'https://api.github.com/user',
...     auth=HTTPBasicAuth('username',
... ))
```

```
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}
```

Though you don't need to be explicit for `requests` provides other methods of auth:

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

[Send Python Tricks »](#)

You can even supply your own authentication mechanism. To do so, you must first create a subclass of `AuthBase`. Then, you implement `__call__()`:

Python

```
import requests
from requests.auth import AuthBase

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

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

    def __call__(self, r):
        """Attach an API token to a custom auth header."""
        r.headers['X-TokenAuth'] = f'{self.token}' # Python 3.6+
        return r

requests.get('https://httpbin.org/get', auth=TokenAuth('12345abcde-token'))
```

Here, your custom `TokenAuth` mechanism receives a token, then includes that token in the `X-TokenAuth` header of your request.

Bad authentication mechanisms can lead to security vulnerabilities, so unless a service requires a custom authentication mechanism for some reason, you'll always want to use a tried-and-true auth scheme like Basic or OAuth.

While you're thinking about security, let's consider dealing with SSL Certificates using `requests`.

[Remove ads](#)

SSL Certificate Verification

Any time the data you are 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 by default. If you want to disable SSL Certificate verification, you can do so:

If you want to disable SSL Certificate verification, you can do so:

Python

```
>>> requests.get('https://api.github.com')
InsecureRequestWarning: Unverified HTTPS request is being made. This
advised. See: https://urllib3.readthedocs.org/en/latest/user/advanced.html#ssl-warnings
InsecureRequestWarning
<Response [200]>
```

```
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}
```

Improve Your Python

...with a fresh **Python Trick** code snippet every couple of days:

[Send Python Tricks »](#)

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.

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 upon 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 things 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 timeout after 1 second. In the second request, the request will timeout after 3.05 seconds.

You can also pass a tuple to `timeout` with the first element being a connect timeout (the time it allows for the client to establish a connection to the server), and the second being a read timeout (the time it will wait on a response once your client has established a connection):

```
>>> requests.get('https://api.github.com', timeout=(2, 5))
<Response [200]>
```

If the request establishes a connection within 2 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

```
import requests
from requests.exceptions import Timeout

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

Your program can catch the `Timeout` exception.

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

[Send Python Tricks »](#)

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, you could use a session:

Python

```
import requests
from getpass import getpass

# By using a context manager, you can ensure the resources used by
# the session will be released after use
with requests.Session() as session:
    session.auth = ('username', getpass())

    # Instead of requests.get(), you'll use session.get()
    response = session.get('https://api.github.com/user')

    # You can inspect the response just like you did before
    print(response.headers)
    print(response.json())
```

Each time you make a request with `session`, once it has been initialized with authentication credentials, the credentials will be persisted.

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 will reuse a connection from the pool rather than establishing a new one.

 [Remove ads](#)

Max Retries

you by default. To apply this functionality, you need to implement a custom [Transport Adapter](#).

Transport Adapters let you define a set of configurations per service you're interacting with. For example, let's say you want all requests to <https://api.github.com> to retry three times before finally raising a `ConnectionError`. You would build a Transport Adapter, set its `max_retries` parameter, and mount it to an existing Session:

Python

```
import requests
from requests.adapters import HTTPAdapter
from requests.exceptions import ConnectionError

github_adapter = HTTPAdapter(max_retries=3)
session = requests.Session()
# Use `github_adapter` for all requests
session.mount('https://api.github.com', github_adapter)

try:
    session.get('https://api.github.com')
except ConnectionError as ce:
    print(ce)
```

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

[Send Python Tricks »](#)

When you mount the `HTTPAdapter`, `github_adapter`, to `session`, `session` will adhere to its configuration for each request to <https://api.github.com>.

Timeouts, Transport Adapters, and sessions are for keeping your code efficient and your application resilient.

Conclusion

You've come a long way in learning about Python's powerful `requests` library.

You're now able 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 using `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.

 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}
```

[Improve Your Python](#)

About Alex Ronquillo



Alex Ronquillo is a Software Engineer at YouVersion. He's an avid Pythonista who is also part of 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}
```

Improve Your Python

...with a fresh **Python Trick** code snippet every couple of days:

Each tutorial at Real Python is created by a community of people. Here are some of the people who worked on this tutorial are:



[Aldren](#)



[Brad](#)



[Joanna](#)

[Send 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}
```

Improve Your Python with **Python Tricks**

Get a short & sweet Python code snippet delivered to your inbox every couple of days:

[» Click here to see examples](#)

What Do You Think?

[Tweet](#) [Share](#) [Email](#)

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

Keep Learning

Related Tutorial Categories: [intermediate](#) [web-dev](#)

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

— FREE Email Series —

Python Tricks

[Improve Your Python](#)

```
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 >>>
10 {'c':
```

Email...

```
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}
```

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

Email Address

[Send Python Tricks »](#)

All Tutorial Topics

[advanced](#) [api](#) [basics](#) [best-practices](#) [community](#) [databases](#) [data-science](#)
[devops](#) [django](#) [docker](#) [flask](#) [front-end](#) [intermediate](#) [machine-learning](#)
[python](#) [testing](#) [tools](#) [web-dev](#) [web-scraping](#)

Python Tricks

realpython.com

Table of Contents

- [Getting Started With requests](#)
- [The GET Request](#)
- [The Response](#)
- [Query String Parameters](#)
- [Request Headers](#)
- [Other HTTP Methods](#)
- [The Message Body](#)
- [Inspecting Your Request](#)
- [Authentication](#)
- [SSL Certificate Verification](#)
- [Performance](#)

Improve Your Python

- Conclusion

[Tweet](#) [Share](#) [Email](#)

 Recommended Video Course

[Making HTTP Requests With Python](#)



High Quality

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

Email Address

[Send Python Tricks »](#)

© 2012–2019 Real Pyt
[Python Tuto](#)

```
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}
```