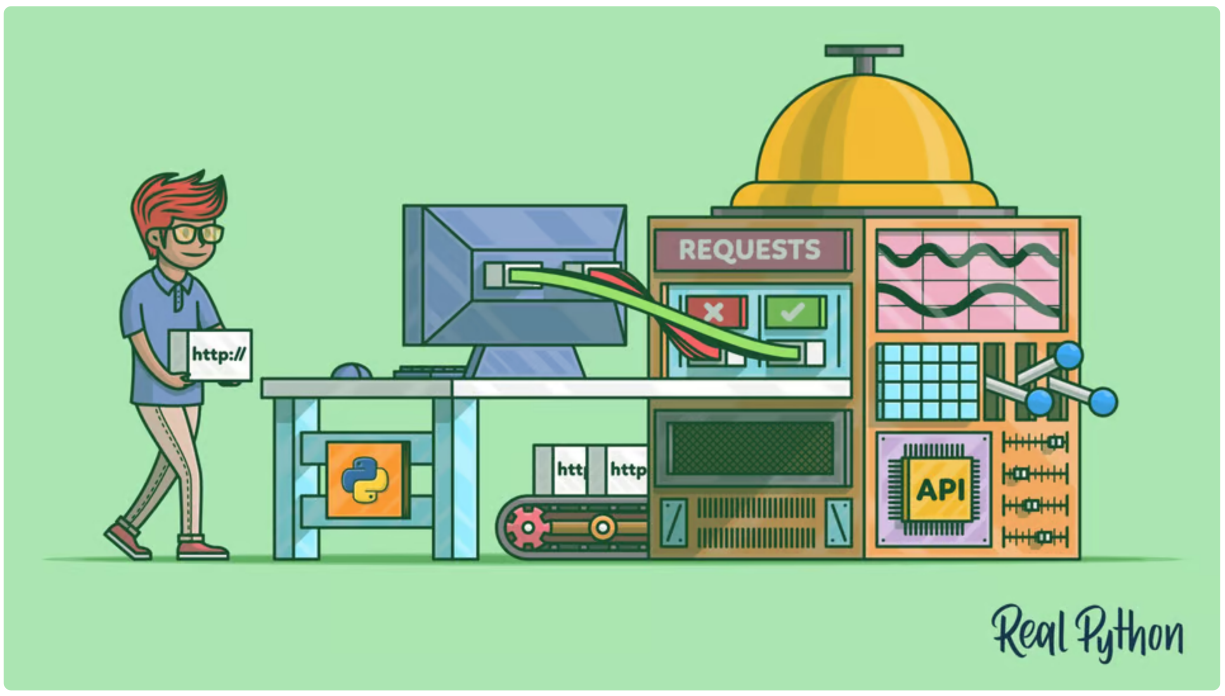

Python’s Requests Library (Guide)

by Alex Ronquillo 31 Comments

intermediate web-dev

Mark as Completed

Tweet Share Email

Table of Contents

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

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

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

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 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

Though I've tried to include as much information as you need to understand the features and examples included in this article, I do assume a *very* basic general knowledge of HTTP. That said, you still may be able to follow along fine anyway.

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

Take the Quiz: Test your knowledge with our interactive "HTTP Requests With the "requests" Library" quiz. Upon completion you will receive a score so you can track your learning progress over time:

Take the Quiz »

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:

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 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 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

```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

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

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 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 and then, if necessary, 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 raise an exception if the request was unsuccessful. You can do this using `.raise_for_status()`:

Python

```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!')
```

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.

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_autho
```

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_author
```

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

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_autho
```

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

You can do a lot with status codes and message bodies. But, if you need more information, like metadata about the response itself, 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 >>>

```
>>> response.headers
{'Server': 'GitHub.com', 'Date': 'Mon, 10 Dec 2018 17:49:54 GMT', 'Conte
```

`.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.

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

```python
import requests

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

# Inspect some attributes of the `requests` repository
json_response = response.json()
repository = json_response['items'][0]
print(f'Repository name: {repository["name"]}')  # Python 3.6+
print(f'Repository description: {repository["description"]}')  # Python
```

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 >>>

```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 >>>

```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

```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()
repository = json_response['items'][0]
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 methods include `POST`, `PUT`, `DELETE`, `HEAD`, `PATCH`, and `OPTIONS`. `requests` provides a method, with a similar signature to `get()`, for each of these HTTP methods:

Python >>>

```python
>>> requests.post('https://httpbin.org/post', data={'key':'value'})
>>> requests.put('https://httpbin.org/put', data={'key':'value'})
>>> requests.delete('https://httpbin.org/delete')
>>> requests.head('https://httpbin.org/get')
>>> requests.patch('https://httpbin.org/patch', data={'key':'value'})
>>> requests.options('https://httpbin.org/get')
```

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 >>>

```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 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':'va
>>> 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`.

Inspecting Your Request

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':'va
>>> 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', getpas
<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`:

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', getpass())
... )
<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`.

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-toke
```

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`.

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 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, you pass `False` to the `verify` parameter of the request function:

Python >>>

```python
>>> requests.get('https://api.github.com', verify=False)
InsecureRequestWarning: Unverified HTTPS request is being made. Adding c
  InsecureRequestWarning)
<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.

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):

Python >>>

```
>>> 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', timeout=1)
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.

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

```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.

Max Retries

When a request fails, you may want your application to retry the same request. However, `requests` will not 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 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

```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 to endpoints that start with thi
session.mount('https://api.github.com', github_adapter)

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

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.

Take the Quiz: Test your knowledge with our interactive “HTTP Requests With the "requests" Library” quiz. Upon completion you will receive a score so you can track your learning progress over time:

Take the Quiz »

Mark as Completed

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.

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

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

Joanna

What Do You Think?

Rate this article:

Tweet Share Share 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 Tutorial Categories: **intermediate** **web-dev**

Recommended Video Course: Making HTTP Requests With Python