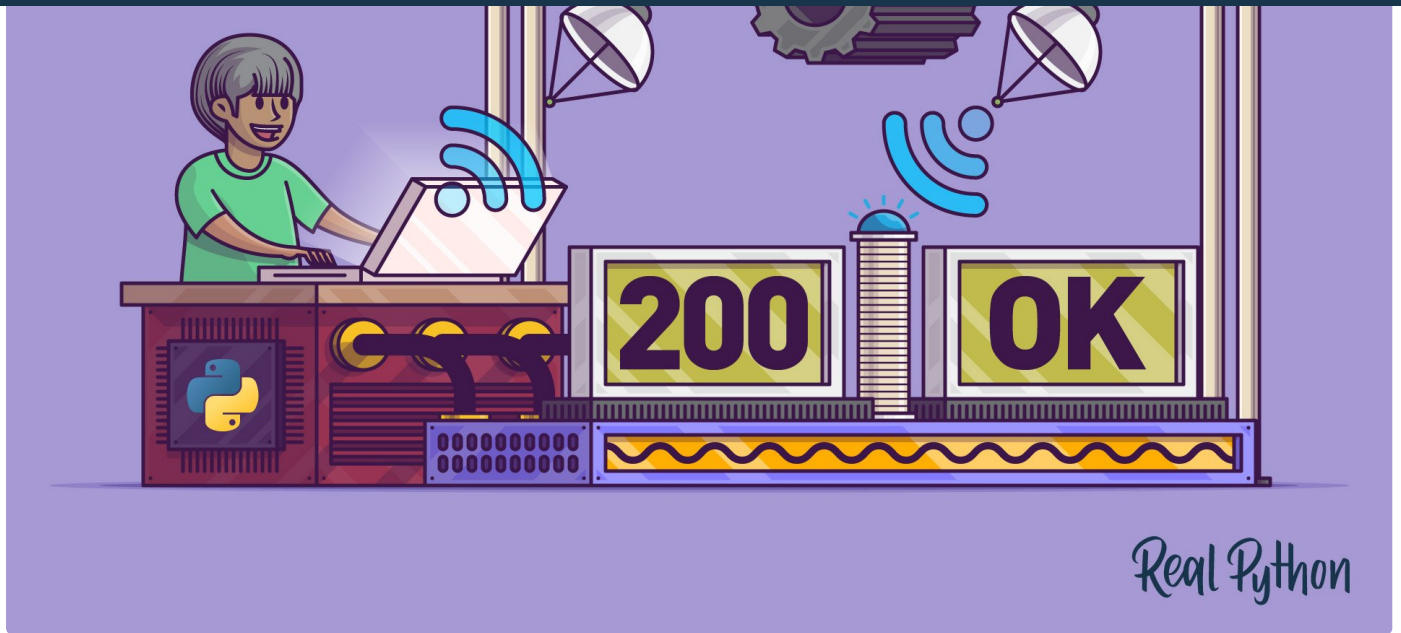




🏠 **Stuck at home?** Enjoy free courses, on us →



Python & APIs: A Winning Combo for Reading Public Data

by Pedro Pagueiro 🕒 Feb 22, 2021 💬 🏷️ api intermediate

Mark as Completed



Tweet

Share

Email

Table of Contents

- [Getting to Know APIs](#)
 - [SOAP vs REST vs GraphQL](#)
 - [requests and APIs: A Match Made in Heaven](#)
- [Calling Your First API Using Python](#)
 - [Endpoints and Resources](#)
 - [Request and Response](#)
 - [Status Codes](#)
 - [HTTP Headers](#)
 - [Response Content](#)
 - [HTTP Methods](#)
 - [Query Parameters](#)
- [Learning Advanced API Concepts](#)
 - [Authentication](#)
 - [Pagination](#)
 - [Rate Limiting](#)
- [Consuming APIs With Python: Practical Examples](#)
 - [Searching and Fetching Trending GIFs](#)
 - [Getting COVID-19 Confirmed Cases Per Country](#)
 - [Searching Google Books](#)

- [Conclusion](#)
- [Further Reading](#)

 Remove ads

Knowing how to consume an API is one of those magical skills that, once mastered, will crack open a whole new world of possibilities, and consuming APIs using Python is a great way to learn such a skill.

A lot of apps and systems you use on a daily basis are connected to an API. From very simple and mundane things, like checking the weather in the morning, to more addictive and time-consuming actions, such as scrolling through your Instagram, TikTok, or Twitter feed, APIs play a central role.

In this tutorial, you'll learn:

- What an **API** is
- How you can **consume APIs** with your Python code
- What the most important **API-related concepts** are
- How to use Python to **read data** available through public APIs

By the end of this tutorial, you'll be able to use Python to consume most APIs you come across. If you're a developer, knowing how to consume APIs with Python will make you much more proficient, especially when it comes to integrating your work with third-party applications.

Note: This tutorial is focused on how to *consume* APIs using Python, not how to build them. For information on building an API with Python, check out [Python REST APIs With Flask, Connexion, and SQLAlchemy](#).

You can download the source code for the examples you'll see in this tutorial by clicking the link below:

Get the Source Code: [Click here to get the source code you'll use](#) to learn about consuming APIs with Python in this tutorial.

 Remove ads

Getting to Know APIs

API stands for **application programming interface**. In essence, an API acts as a communication layer, or as the name says, an interface, that allows different systems to talk to each other without having to understand exactly what each other does.

APIs can come in many forms or shapes. They can be operating system APIs, used for actions like turning on your camera and audio for joining a Zoom call. Or they can be web APIs, used for web-focused actions such as liking

images on your Instagram or fetching the latest tweets.

No matter the type, all APIs function mostly the same way. You usually make a **request** for information or data, and the API returns a **response** with what you requested. For example, every time you open Twitter or scroll down your Instagram feed, you're basically making a request to the API behind that app and getting a response in return. This is also known as **calling** an API.

In this tutorial you'll focus more on the high-level APIs that communicate across networks, also called **web APIs**.

SOAP vs REST vs GraphQL

Even though some of the examples mentioned above are geared toward newer platforms or apps, web APIs have been around for quite a long time. In the late 1990s and early 2000s, two different design models became the norm in exposing data publicly:

1. **SOAP (Simple Object Access Protocol)** is typically associated with the enterprise world, has a stricter contract-based usage, and is mostly designed around actions.
2. **REST (Representational State Transfer)** is typically used for public APIs and is ideal for fetching data from the web. It's much lighter and closer to the HTTP specification than SOAP.

Nowadays, there's a new kid in town: [GraphQL](#). Created by Facebook, GraphQL is a very flexible query language for APIs, where the clients decide exactly what they want to fetch from the server instead of the server deciding what to send.

If you want to learn more about the differences between these three design models, then here are a few good resources:

- [What is SOAP?](#)
- [What is REST?](#)
- [API 101: SOAP vs. REST](#)
- [Introduction to GraphQL](#)
- [Comparing API Architectural Styles: SOAP vs REST vs GraphQL vs RPC](#)

Even though GraphQL is on the rise and is being adopted by bigger and bigger companies, including [GitHub](#) and [Shopify](#), the truth is that the majority of public APIs are still REST APIs. Therefore, for the purpose of this tutorial, you'll learn only about REST APIs and how to consume them using Python.

requests and APIs: A Match Made in Heaven

When consuming APIs with Python, there's only one library you need: [requests](#). With it, you should be able to do most, if not all, of the actions required to consume any public API.

You can install requests by running the following command in your console:

Shell

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

To follow the code examples in this tutorial, make sure you're using Python 3.8.1 and [requests 2.24.0](#) or higher.

Calling Your First API Using Python

Enough talking—let's start making requests

Enough talking—it's time to make your first API call! For the first example, you'll be calling a popular API for generating [random user data](#).

Throughout the tutorial, you'll see new APIs introduced in alert blocks like the one below. It's a convenient way for you to scroll through afterward and quickly spot all the new APIs you learned about.

Random User Generator API: This is a great tool for [generating random user data](#). You can use it to generate any number of random users and associated data, and you can also specify gender, nationality, and many other filters that can be really helpful when testing apps or, in this case, APIs.

The only thing you need to start with the Random User Generator API is to know which URL to call it with. For this example, the URL to use is `https://randomuser.me/api/`, and this is the tiniest API call you can make:

Python

>>>

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

In this small example, you [import](#) the requests library and then fetch (or get) data from the URL for the Random User Generator API. But you don't actually see any of the data returned. What you get instead is a Response [200], which in API terms means everything went OK.

If you want to see the actual data, then you can use `.text` from the returned Response object:

Python

>>>

```
>>> import requests
>>> response = requests.get("https://randomuser.me/api/")
>>> response.text
'{"results":[{"gender":"female",
"name":{"title":"Ms","first":"Isobel","last":"Wang"}...}'
```

That's it! That's the very basics of API consumption. You managed to fetch your first random user from the Random User Generator API using Python and the requests library.

 Remove ads

Endpoints and Resources

As you saw above, the first thing you need to know for consuming an API is the API URL, typically called the **base URL**. The base URL structure is no different from the URLs you use for browsing Google, YouTube, or Facebook, though it usually contains the word `api`. This is not mandatory, just more of a rule of thumb.

For example, here are the base URLs for a few well-known API players:

- `https://api.twitter.com`
- `https://api.github.com`

- <https://api.stripe.com>

As you can see, all of the above start with `https://api` and include the remaining official domain, such as `.twitter.com` or `.github.com`. There's no specific standard for how the API base URL should look, but it's quite common for it to mimic this structure.

If you try opening any of the above links, then you'll notice that most of them will return an error or ask for credentials. That's because APIs sometimes require authentication steps before you can use them. You'll learn more about this [a bit later](#) in the tutorial.

TheDogAPI: This API is quite fun but also a really good example of a well-done API with great [documentation](#). With it, you can fetch the different dog breeds and some images, but if you register, you can also cast votes on your favorite dogs.

Next, using the just-introduced [TheDogAPI](#), you'll try to make a basic request to see how it may differ from the Random User Generator API you tried above:

Python

>>>

```
>>> import requests
>>> response = requests.get("https://api.thedogapi.com/")
>>> response.text
'{"message": "The Dog API"}'
```

In this case, when calling the base URL, you get this generic message saying `The Dog API`. This is because you're calling the base URL, which is typically used for very basic information about an API, not the real data.

Calling the base URL alone isn't a lot of fun, but that's where endpoints come in handy. An **endpoint** is a part of the URL that specifies what **resource** you want to fetch. Well-documented APIs usually contain an **API reference**, which is extremely useful for knowing the exact endpoints and resources an API has and how to use them.

You can check [the official documentation](#) to learn more about how to use TheDogAPI and what endpoints are available. In there, you'll find a [/breeds endpoint](#) that you can use to fetch all the available breed resources or objects.

If you scroll down, then you'll find the [Send a Test Request](#) section, where you'll see a form like the following:

Send a Test Request

▶ Send

GET

`https://api.thedogapi.com/v1/breeds`

Settings

Headers [1]

Query [3]

Code Generation

Send requests directly from the browser (CORS must be enabled) ☐

▼ `$$env` 1 variable not set

`x-api-key`

value

This is something that you'll see in many API documentations: a way for you to quickly test the API directly from the documentation page. In this case, you can click *Send* to quickly get the result of calling that endpoint. *Et voilà*, you just called an API without having to write any code for it.

Now, give it a try in code locally using the [breeds endpoint](#) and some of the API knowledge you already have:

Python

>>>

```
>>> response = requests.get("https://api.thedogapi.com/v1/breeds")
>>> response.text
'{"weight":{"imperial":"6 - 13","metric":"3 - 6"},"height": ...}'
```

There you go, your first breed listing using the dog API!

If you're a cat person, don't fret. There's an API for you, too, with the same endpoint but a different base URL:

Python

>>>

```
>>> response = requests.get("https://api.thecatapi.com/v1/breeds")
>>> response.text
'[{... "id": "abys", "name": "Abyssinian"}]'
```

I bet you're already thinking about different ways you can use these APIs to make some cute side project, and that's the great thing about APIs. Once you start using them, there's nothing stopping you from turning a hobby or passion into a fun little project.

Before you move forward, one thing you need to know about endpoints is the difference between `http://` and `https://`. In a nutshell, HTTPS is the encrypted version of HTTP, making all traffic between the client and the server much safer. When consuming public APIs, you should definitely stay away from sending any private or sensitive information to `http://` endpoints and use only those APIs that provide a secure `https://` base URL.

For more information on why it's important to stick to HTTPS when online browsing, check out [Exploring HTTPS With Python](#).

In the next section, you'll dig a bit further into the main components of an API call.

 Remove ads

Request and Response

As you very briefly read above, all interactions between a client—in this case your Python console—and an API are split into a request and a response:

- **Requests** contain relevant data regarding your API request call, such as the base URL, the endpoint, the method used, the headers, and so on.
- **Responses** contain relevant data returned by the server, including the data or content, the status code, and the headers.

Using TheDogAPI again, you can drill down a bit more into what exactly is inside the `Request` and `Response` objects:

```
>>> response = requests.get("https://api.thedogapi.com/v1/breeds")
>>> response
<Response [200]>
>>> response.request
<PreparedRequest [GET]>

>>> request = response.request
>>> request.url
'https://api.thedogapi.com/v1/breeds'
>>> request.path_url
'/v1/breeds'
>>> request.method
'GET'
>>> request.headers
{'User-Agent': 'python-requests/2.24.0', 'Accept-Encoding': 'gzip, deflate',
'Accept': '*/*', 'Connection': 'keep-alive'}

>>> response
<Response [200]>
>>> response.text
'[{"weight":{"imperial":"6 - 13","metric":"3 - 6"},
"height":{"imperial":"9 - 11.5","metric":"23 - 29"},"id":1,
"name":"Affenpinscher", ...}]'
>>> response.status_code
200
>>> response.headers
{'Cache-Control': 'post-check=0, pre-check=0', 'Content-Encoding': 'gzip',
'Content-Type': 'application/json; charset=utf-8',
'Date': 'Sat, 25 Jul 2020 17:23:53 GMT'...}
```

The example above shows you a few of the most important attributes available for Request and Response objects.

You'll learn more about some of these attributes in this tutorial, but if you want to dig even further, then you can check Mozilla's documentation on [HTTP messages](#) for a more in-depth explanation of each attribute.

Status Codes

Status codes are one of the most important pieces of information to look for in any API response. They tell you if your request was successful, if it's missing data, if it's missing credentials, [and so on](#).

With time, you'll recognize the different status codes without help. But for now, here's a list with some of the most common status codes you'll find:

| Status code | Description |
|---------------------------|---|
| 200 OK | Your request was successful! |
| 201 Created | Your request was accepted and the resource was created. |
| 400 Bad Request | Your request is either wrong or missing some information. |
| 401 Unauthorized | Your request requires some additional permissions. |
| 404 Not Found | The requested resource does not exist. |
| 405 Method Not Allowed | The endpoint does not allow for that specific HTTP method. |
| 500 Internal Server Error | Your request wasn't expected and probably broke something on the server side. |

You saw 200 OK earlier in the examples you executed, and you might even recognize 404 Not Found from browsing the web.

Fun fact: Companies tend to use 404 error pages for private jokes or pure fun, like these examples below:

- [Mantra Labs](#)
- [Gymbox](#)
- [Pixar](#)
- [Slack](#)

In the API world, though, developers have limited space in the response for this kind of fun. But they make up for it in other places, like the HTTP headers. You'll see some examples soon enough!

You can check the status of a response using `.status_code` and `.reason`. The requests library also prints the status code in the representation of the Response object:

Python

>>>

```
>>> response = requests.get("https://api.thedogapi.com/v1/breeds")
>>> response
<Response [200]>
>>> response.status_code
200
>>> response.reason
'OK'
```

The request above returns 200, so you can consider it a successful request. But now have a look at a failing request triggered when you include a typo in the endpoint `/breedz`:

Python

>>>

```
>>> response = requests.get("https://api.thedogapi.com/v1/breedz")
>>> response
<Response [404]>
>>> response.status_code
404
>>> response.reason
'Not Found'
```

As you can see, the /breedz endpoint doesn't exist, so the API returns a 404 Not Found status code.

You can use these status codes to quickly see if your request needs to be changed or if you should check the documentation again for any typos or missing pieces.

HTTP Headers

HTTP headers are used to define a few parameters governing requests and responses:

| HTTP Header | Description |
|----------------|--|
| Accept | What type of content the client can accept |
| Content-Type | What type of content the server will respond with |
| User-Agent | What software the client is using to communicate with the server |
| Server | What software the server is using to communicate with the client |
| Authentication | Who is calling the API and what credentials they have |

There are many other headers that you can find when inspecting a request or response. Have a look at [Mozilla's extended list](#) if you're curious about the specific use for each of them.

To inspect the headers of a response, you can use `response.headers`:

Python

>>>

```
>>> response = requests.get("https://api.thedogapi.com/v1/breeds/1")
>>> response.headers
{'Content-Encoding': 'gzip',
 'Content-Type': 'application/json; charset=utf-8',
 'Date': 'Sat, 25 Jul 2020 19:52:07 GMT'...}
```

To do the same with the request headers, you can use `response.request.headers` since request is an attribute of the Response object:

Python

>>>

```
>>> response = requests.get("https://api.thedogapi.com/v1/breeds/1")
>>> response.request.headers
{'User-Agent': 'python-requests/2.24.0',
 'Accept-Encoding': 'gzip, deflate', 'Accept': '*/*',
 'Connection': 'keep-alive'}
```

In this case, you don't define any specific headers when you make the request, so the default headers are returned.

Custom Headers

Another standard that you might come across when consuming APIs is the use of custom headers. These usually start with X-, but they're not required to. API developers typically use custom headers to send or request additional custom information from clients.

Fun fact: A few companies go to extra lengths to be funny and innovative, using HTTP headers in a way they weren't meant to be used, such as to solicit job applications.

You can use a [dictionary](#) to define headers, and you can send them along with your request using the `headers` parameter of `.get()`.

For example, say you want to send some request ID to the API server, and you know you can do that using `X-Request-Id`:

```
Python >>>
>>> headers = {"X-Request-Id": "<my-request-id>"}
>>> response = requests.get("https://example.org", headers=headers)
>>> response.request.headers
{'User-Agent': 'python-requests/2.24.0', 'Accept-Encoding': 'gzip, deflate',
'Accept': '*/.*', 'Connection': 'keep-alive',
'X-Request-Id': '<my-request-id>'}
```

If you go through the `request.headers` dictionary, then you'll find `X-Request-Id` right at the end, among a few other headers that come by default with any API request.

There are many useful headers a response might have, but one of the most important ones is `Content-Type`, which defines the kind of content returned in the response.

Content-Type

These days, most APIs use [JSON](#) as the default content type, but you might need to use an API that returns XML or other media types, such as images or video. In that case, the content type will differ.

If you look back to one of the previous examples using TheDogAPI and try to inspect the `Content-Type` header, then you'll notice how it was defined as `application/json`:

```
Python >>>
>>> response = requests.get("https://api.thedogapi.com/v1/breeds/1")
>>> response.headers.get("Content-Type")
'application/json; charset=utf-8'
```

Apart from the specific type of content (in this case `application/json`), the header might also return the specified [encoding](#) for the response content.

PlaceGOAT API: This is a very silly API that returns [images of goats](#) in different sizes that you can use as placeholder images in your website.

If, for example, you try to fetch an image of a goat from the [PlaceGOAT API](#), then you'll notice that the content type is no longer `application/json`, but instead it's defined as `image/jpeg`:

Python

>>>

```
>>> response = requests.get("http://placegoat.com/200/200")
>>> response
<Response [200]>
>>> response.headers.get("Content-Type")
'image/jpeg'
```

In this case, the `Content-Type` header states that the returned content is a JPEG image. You'll learn how to view this content in the next section.

The `Content-Type` header is very important for you to know how to handle a response and what to do with its content. There are [hundreds](#) of other acceptable content types, including audio, video, fonts, and more.

 Remove ads

Response Content

As you just learned, the type of content you find in the API response will vary according to the `Content-Type` header. To properly read the response contents according to the different `Content-Type` headers, the `requests` package comes with a couple of different `Response` attributes you can use to manipulate the response data:

- `.text` returns the response contents in [Unicode](#) format.
- `.content` returns the response contents in [bytes](#).

You already used the `.text` attribute above. But for some specific types of data, like images and other nontextual data, using `.content` is typically a better approach, even if it returns a very similar result to `.text`:

Python

>>>

```
>>> response = requests.get("https://api.thedogapi.com/v1/breeds/1")
>>> response.headers.get("Content-Type")
'application/json; charset=utf-8'
>>> response.content
b'{"weight":{"imperial":"6 - 13", "metric":"3 - 6"}}...'
```

As you can see, there isn't a big difference between `.content` and the previously used `.text`.

However, by looking at the response's `Content-Type` header, you can see the content is `application/json`, a JSON object. For that kind of content, the `requests` library includes a specific `.json()` method that you can use to immediately convert the API bytes response into a [Python data structure](#):

Python

>>>

```
>>> response = requests.get("https://api.thedogapi.com/v1/breeds/1")
>>> response.headers.get("Content-Type")
'application/json; charset=utf-8'
>>> response.json()
{'weight': {'imperial': '6 - 13', 'metric': '3 - 6'},
 'height': {'imperial': '9 - 11.5', 'metric': '23 - 29'}
...}
>>> response.json()["name"]
'Affenpinscher'
```

As you can see, after executing `response.json()`, you get a dictionary that you're able to use as you'd use any other dictionary in Python.

Now, looking back at the recent example you ran using the PlaceGOAT API, try to fetch that same goat image and have a look at its content:

Python

>>>

```
>>> response = requests.get("http://placegoat.com/200/200")
>>> response
<Response [200]>
>>> response.headers.get("Content-Type")
'image/jpeg'
>>> response.content
b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x01\x00H...'
```

In this case, because you're requesting an image, `.content` isn't very helpful. In fact, it's nearly impossible to understand. However, you know this is a JPEG image, so you can try storing it into a [file](#) and see what happens:

Python

>>>

```
>>> response = requests.get("http://placegoat.com/200/200")
>>> response
<Response [200]>
>>> response.headers.get("Content-Type")
'image/jpeg'
>>> file = open("goat.jpeg", "wb")
>>> file.write(response.content)
>>> file.close()
```

Now if you open the folder you're working from, then you'll find a `goat.jpeg` file, which is a random image of a goat that you just fetched using an API. Isn't that amazing?

HTTP Methods

When calling an API, there are a few different methods, also called **verbs**, that you can use to specify what action you want to execute. For example, if you wanted to fetch some data, you'd use the method GET, and if you wanted to create some data, then you'd use the method POST.

When purely consuming data using APIs, you'll typically stick to GET requests, but here's a list of the most common methods and their typical use case:

| HTTP Method | Description | Requests method |
|-------------|------------------------------|--------------------------------|
| POST | Create a new resource. | <code>requests.post()</code> |
| GET | Read an existing resource. | <code>requests.get()</code> |
| PUT | Update an existing resource. | <code>requests.put()</code> |
| DELETE | Delete an existing resource. | <code>requests.delete()</code> |

These four methods are typically referred to as **CRUD operations** as they allow you to **create**, **read**, **update** and **delete** resources.

Note: There's an additional PATCH method that's also associated with CRUD operations, but it's slightly less common than the four above. It's used to make partial modifications instead of completely replacing a resource using PUT.

You can read a bit more about the [differences between PUT and PATCH](#) to understand their different needs.

If you're curious about the remaining HTTP methods, or if you just want to learn a bit more about those already mentioned, then have a look through [Mozilla's documentation](#).

Until now, you've only used `.get()` to fetch data, but you can use the `requests` package for all the other HTTP methods as well:

Python

>>>

```
>>> requests.post("https://api.thedogapi.com/v1/breeds/1")
>>> requests.get("https://api.thedogapi.com/v1/breeds/1")
>>> requests.put("https://api.thedogapi.com/v1/breeds/1")
>>> requests.delete("https://api.thedogapi.com/v1/breeds/1")
```

If you try these on your console, then you'll notice that most of them will return a 405 Method Not Allowed [status code](#). That's because not all endpoints will allow for POST, PUT, or DELETE methods. Especially when you're reading data using public APIs, you'll find that most APIs will only allow GET requests since you're not allowed to create or change the existing data.

 Remove ads

Query Parameters

Sometimes when you call an API, you get a ton of data that you don't need or want. For example, when calling TheDogAPI's `/breeds` endpoint, you get a lot of information about a given breed. But in some cases, you might want to extract only certain information about a given breed. That's where query parameters come in!

You might have seen or used query parameters when browsing online. For example when watching a YouTube

video, you have a URL like `https://www.youtube.com/watch?v=aL5GK2LVMWI`. The `v=` in the URL is what you call a **query parameter**. It typically comes after the base URL and endpoint.

To add a query parameter to a given URL, you have to add a question mark (`?`) before the first query parameter. If you want to have multiple query parameters in your request, then you can split them with an ampersand (`&`).

The same YouTube URL above with multiple query parameters would look like this: `https://www.youtube.com/watch?v=aL5GK2LVMWI&t=75`.

In the API world, query parameters are used as filters you can send with your API request to further narrow down the responses. For example, going back to the Random User Generator API, you know how to generate a random user:

Python

>>>

```
>>> requests.get("https://randomuser.me/api/").json()
{'results': [{'gender': 'male', 'name':
{'title': 'Mr', 'first': 'Silvijn', 'last': 'Van Bekkum'},
'location': {'street': {'number': 2480, 'name': 'Hooijengastrjitte'},
'city': 'Terherne', 'state': 'Drenthe',
'country': 'Netherlands', 'postcode': 59904...}]}
```

However, let's say you specifically want to generate only random female users. According to the [documentation](#), you can use the query parameter `gender=` for that:

Python

>>>

```
>>> requests.get("https://randomuser.me/api/?gender=female").json()
{'results': [{'gender': 'female', 'name':
{'title': 'Mrs', 'first': 'Marjoleine', 'last': 'Van Huffelen'},
'location': {'street': {'number': 8993, 'name': 'De Teebus'},
'city': 'West-Terschelling', 'state': 'Limburg',
'country': 'Netherlands', 'postcode': 24241...}]}
```

That's great! Now let's say you want to generate only female users from Germany. Again, looking through the documentation, you find a section on [nationality](#), and you can use the query parameter `nat=` for that:

Python

>>>

```
>>> requests.get("https://randomuser.me/api/?gender=female&nat=de").json()
{'results': [{'gender': 'female', 'name':
{'title': 'Ms', 'first': 'Marita', 'last': 'Hertwig'},
'location': {'street': {'number': 1430, 'name': 'Waldstraße'},
'city': 'Velden', 'state': 'Rheinland-Pfalz',
'country': 'Germany', 'postcode': 30737...}]}
```

Using query parameters, you can start fetching more specific data from an API, making the whole experience a bit more tailored to your needs.

To avoid having to rebuild the URL over and over again, you can use the `params` attribute to send in a dictionary of all query parameters to append to a URL:

Python

>>>

```
>>> query_params = {"gender": "female", "nat": "de"}
>>> requests.get("https://randomuser.me/api/", params=query_params).json()
{'results': [{'gender': 'female', 'name':
{'title': 'Ms', 'first': 'Janet', 'last': 'Weyer'},
'location': {'street': {'number': 2582, 'name': 'Meisenweg'},
'city': 'Garding', 'state': 'Mecklenburg-Vorpommern',
'country': 'Germany', 'postcode': 56953...}]}
```

You can apply the above to any other API you like. If you go back to TheDogAPI, the documentation has a way for you to [filter the breeds endpoint](#) to return only the breeds that match a specific name. For example, if you wanted to look for the Labradoodle breed, then you could do that with the query parameter `q`:

Python

>>>

```
>>> query_params = {"q": "labradoodle"}
>>> endpoint = "https://api.thedogapi.com/v1/breeds/search"
>>> requests.get(endpoint, params=query_params).json()
[{'weight': {'imperial': '45 - 100', 'metric': '20 - 45'},
'height': {'imperial': '14 - 24', 'metric': '36 - 61'},
'id': 148, 'name': 'Labradoodle', 'breed_group': 'Mixed'...}]
```

There you have it! By sending the query parameter `q` with the value `labradoodle`, you're able to filter all breeds that match that specific value.

Tip: When you're reusing the same endpoint, it's a best practice to define it as a [variable](#) at the top of your code. This will make your life easier when interacting with an API over and over again.

With the help of query parameters, you're able to further narrow your requests and specify exactly what you're looking for. Most APIs you'll find online will have some sort of query parameters that you can use to filter data. Remember to look through the documentation and API reference to find them.

Learning Advanced API Concepts

Now that you have a good understanding of the basics of API consumption using Python, there are a few more advanced topics that are worth touching upon, even if briefly, such as [authentication](#), [pagination](#), and [rate limiting](#).

 Remove ads

Authentication

API authentication is perhaps the most complex topic covered in this tutorial. Even though a lot of public APIs are free and completely public, an even bigger number of APIs are available behind some form of authentication. There are numerous APIs that require authentication, but here are a few good examples:

- [GitHub API](#)
- [Twitter API](#)

- [Instagram API](#)

Authentication approaches range from the simplistic and straightforward, like those using API keys or Basic Authentication, to much more complex and safer techniques, like OAuth.

Typically, calling an API without credentials or with the wrong ones will return a 401 Unauthorized or 403 Forbidden status code.

API Keys

The most common level of authentication is the **API key**. These keys are used to identify you as an API user or customer and to trace your use of the API. API keys are typically sent as a request header or as a query parameter.

NASA APIs: One of the coolest collections of publicly available APIs is the one provided by [NASA](#). You can find APIs to fetch the [astronomy picture of the day](#) or pictures taken by the [Earth Polychromatic Imaging Camera \(EPIC\)](#), among others.

For this example, you'll have a go at NASA's [Mars Rover Photo API](#), and you'll fetch pictures taken on July 1, 2020. For testing purposes, you can use the DEMO_KEY API key that NASA provides by default. Otherwise, you can quickly generate your own by going to NASA's [main API page](#) and clicking *Get Started*.

You can add the API key to your request by appending the `api_key=` query parameter:

Python

>>>

```
>>> endpoint = "https://api.nasa.gov/mars-photos/api/v1/rovers/curiosity/photos"
>>> # Replace DEMO_KEY below with your own key if you generated one.
>>> api_key = "DEMO_KEY"
>>> query_params = {"api_key": api_key, "earth_date": "2020-07-01"}
>>> response = requests.get(endpoint, params=query_params)
>>> response
<Response [200]>
```

So far, so good. You managed to make an authenticated request to NASA's API and to get back a 200 OK response.

Now have a look at the Response object and try to extract some pictures from it:


```
>>> response.json()
{'photos': [{'id': 754118,
  'sol': 2809,
  'camera': {'id': 20,
    'name': 'FHAZ',
    'rover_id': 5,
    'full_name': 'Front Hazard Avoidance Camera'},
  'img_src': 'https://mars.nasa.gov/msl-raw-images/...JPG',
  'earth_date': '2020-07-01',
  'rover': {'id': 5,
    'name': 'Curiosity',
    'landing_date': '2012-08-06',
    'launch_date': '2011-11-26',
    'status': 'active'}},
  ...
]}
>>> photos = response.json()["photos"]
>>> print(f"Found {len(photos)} photos")
Found 12 photos
>>> photos[4]["img_src"]
'https://mars.nasa.gov/msl-raw-images/proj/msl/redops/ods/surface/sol/02809/opgs/edr/rcam/RRB_646869036EDR_F0810628RHAZ0033'
```

Using `.json()` to convert the response to a Python dictionary and then fetching the `photos` field from the response, you're able to iterate through all Photo objects and even fetch a specific photo's image URL. If you open that URL in your browser, then you'll see the following picture of Mars taken by one of the [Mars rovers](#):



Mars Rover API Picture

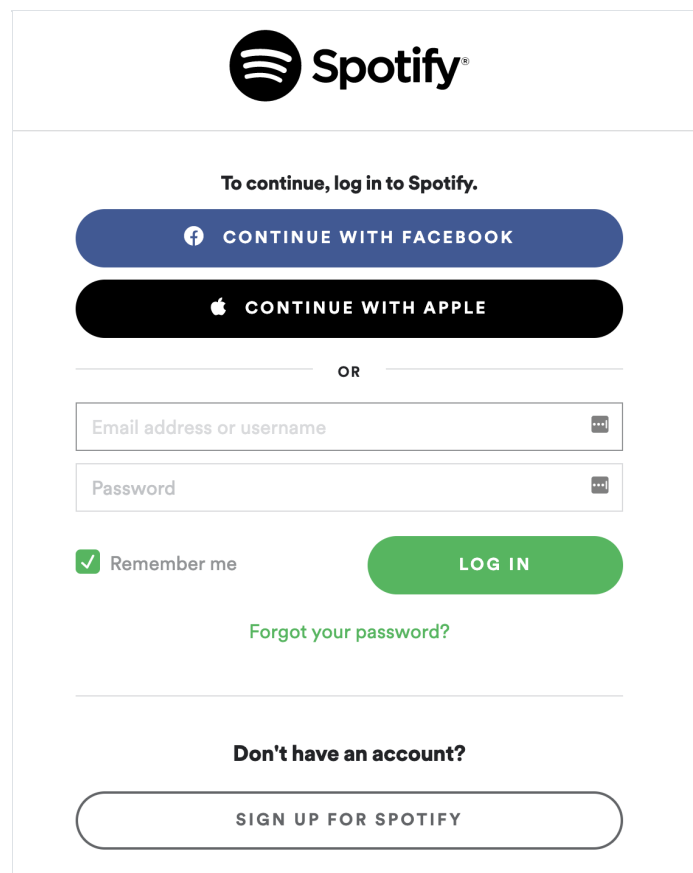
For this example, you picked a specific `earth_date` (2020-07-01) and then a specific photo from the response dictionary (4). Before moving forward, try changing the date or fetching pictures from a different [camera](#) to see how it changes the end result.

OAuth: Getting Started

Another very common standard in API authentication is [OAuth](#). You'll learn only the essentials of OAuth in this

tutorial since it's a very broad topic.

Even if you weren't aware that it was part of OAuth, you may have seen and used the OAuth flow multiple times. Every time an app or platform has a *Login With* or *Continue With* option, that's the starting point of an OAuth flow:

The image shows the Spotify login interface. At the top is the Spotify logo. Below it, the text "To continue, log in to Spotify." is displayed. There are two large, rounded buttons: a blue one with the Facebook logo and the text "CONTINUE WITH FACEBOOK", and a black one with the Apple logo and the text "CONTINUE WITH APPLE". Below these is a horizontal line with the word "OR" in the center. Underneath the line are two input fields: "Email address or username" and "Password", each with a small icon on the right. Below the input fields is a checkbox labeled "Remember me" with a green checkmark, and a green "LOG IN" button. Below the "LOG IN" button is a green link that says "Forgot your password?". At the bottom, there is a horizontal line, followed by the text "Don't have an account?" and a rounded button that says "SIGN UP FOR SPOTIFY".

Example OAuth Login Buttons: Spotify

Here's a step-by-step breakdown of what will happen if you click *Continue With Facebook*:

1. The Spotify app will ask the Facebook API to start an authentication flow. To do this, the Spotify app will send its application ID (`client_id`) and a URL (`redirect_uri`) to redirect the user after success or error.
2. You'll be redirected to the Facebook website and asked to log in with your credentials. The Spotify app won't see or have access to these credentials. *This is the most important benefit of OAuth.*
3. Facebook will show you all the data the Spotify app is requesting from your profile and ask you to accept or reject sharing that data.
4. If you accept giving Spotify access to your data, then you'll be redirected back to the Spotify app, already logged in.

When going through step 4, Facebook will provide Spotify with a special credential (`access_token`) that can be used repeatedly to fetch your information. This specific Facebook login token is valid for sixty days, but other apps might have different expiration periods. If you're curious, then Facebook has a [settings page](#) that you can check to see which apps have been given your Facebook access token.

Now, from a more technical standpoint, here are the things you need to know when consuming APIs using OAuth:

- You need to create an application that will have an ID (`app_id` or `client_id`) and a secret (`app_secret` or `client_secret`).
- You need to have a redirect URL (`redirect_uri`), which the API will use to send information to you.

- You'll get a code as the result of the authentication, which you need to exchange for an access token.

There are a few variations to the above, but generally speaking, most OAuth flows will have steps similar to these.

Tip: When you're just testing things out and you need some sort of redirect URL to get a code, you can use a service called [httpbin](https://httpbin.org/).

More specifically, you can use <https://httpbin.org/anything> as a redirect URL, as it'll simply output whatever it gets as an input. You can test it yourself by navigating to that URL.

Next, you'll dive into an example using the GitHub API!

OAuth: A Practical Example

As you saw above, the first thing you need to do is create an application. There's a great step-by-step explanation on how to do this in the [GitHub documentation](#) that you can follow. The only thing to keep in mind is to use the <https://httpbin.org/anything> URL mentioned above for the *Authorization callback URL* field.

GitHub API: You can use the [GitHub API](#) for a lot of different use cases, such as getting a list of repositories you're a part of, getting a list of followers you have, and much more.

Once you've created your app, copy and paste the `Client_ID` and `Client_Secret`, together with your selected redirect URL, into a Python file called `github.py`:

Python

```
import requests

# REPLACE the following variables with your Client ID and Client Secret
CLIENT_ID = "<REPLACE_WITH_CLIENT_ID>"
CLIENT_SECRET = "<REPLACE_WITH_CLIENT_SECRET>"

# REPLACE the following variable with what you added in the
# "Authorization callback URL" field
REDIRECT_URI = "<REPLACE_WITH_REDIRECT_URI>"
```

Now that you have all the important variables in place, you need to be able to create a link to redirect the user to their GitHub account, as explained in the [GitHub documentation](#):

Python

```
def create_oauth_link():
    params = {
        "client_id": CLIENT_ID,
        "redirect_uri": REDIRECT_URI,
        "scope": "user",
        "response_type": "code",
    }

    endpoint = "https://github.com/login/oauth/authorize"
    response = requests.get(endpoint, params=params)
    url = response.url
    return url
```

In this piece of code, you first define the required parameters that the API expects and then call the API using the requests package and .get().

When you make the request to the /login/oauth/authorize endpoint, the API will automatically redirect you to the GitHub website. In that case, you want to fetch the url parameter from the response. This parameter contains the exact URL that GitHub is redirecting you to.

The next step in the authorization flow is to exchange the code you get for an access token. Again, following the steps in [GitHub's documentation](#), you can make a method for it:

Python

```
def exchange_code_for_access_token(code=None):
    params = {
        "client_id": CLIENT_ID,
        "client_secret": CLIENT_SECRET,
        "redirect_uri": REDIRECT_URI,
        "code": code,
    }

    headers = {"Accept": "application/json"}
    endpoint = "https://github.com/login/oauth/access_token"
    response = requests.post(endpoint, params=params, headers=headers).json()
    return response["access_token"]
```

Here, you make a POST request to exchange the code for an access token. In this request, you have to send your CLIENT_SECRET and code so that GitHub can validate that this specific code was initially generated by your application. Only then will the GitHub API generate a valid access token and return it to you.

Now you can add the following to your file and try running it:

Python

```
link = create_oauth_link()
print(f"Follow the link to start the authentication with GitHub: {link}")
code = input("GitHub code: ")
access_token = exchange_code_for_access_token(code)
print(f"Exchanged code {code} with access token: {access_token}")
```

If everything goes according to plan, then you should be rewarded with a valid access token that you can use to make calls to the GitHub API, impersonating the authenticated user.

Now try adding the following code to fetch your user profile using the [User API](#) and to print your name, username, and number of private repositories:

Python

```
def print_user_info(access_token=None):
    headers = {"Authorization": f"token {access_token}"}
    endpoint = "https://api.github.com/user"
    response = requests.get(endpoint, headers=headers).json()
    name = response["name"]
    username = response["login"]
    private_repos_count = response["total_private_repos"]
    print(
        f"{name} ({username}) | private repositories: {private_repos_count}"
    )
```

Now that you have a valid access token, you need to send it on all your API requests using the `Authorization` header. The response to your request will be a Python dictionary containing all the user information. From that dictionary, you want to fetch the fields `name`, `login`, and `total_private_repos`. You can also print the `response` variable to see what other fields are available.

Alright, that should be it! The only thing left to do is to put it all together and try it out:

Python

```
1 import requests
2
3 # REPLACE the following variables with your Client ID and Client Secret
4 CLIENT_ID = "<REPLACE_WITH_CLIENT_ID>"
5 CLIENT_SECRET = "<REPLACE_WITH_CLIENT_SECRET>"
6
7 # REPLACE the following variable with what you added in
8 # the "Authorization callback URL" field
9 REDIRECT_URI = "<REPLACE_WITH_REDIRECT_URI>"
10
11 def create_oauth_link():
12     params = {
13         "client_id": CLIENT_ID,
14         "redirect_uri": REDIRECT_URI,
15         "scope": "user",
16         "response_type": "code",
17     }
18     endpoint = "https://github.com/login/oauth/authorize"
19     response = requests.get(endpoint, params=params)
20     url = response.url
21     return url
22
23 def exchange_code_for_access_token(code=None):
24     params = {
25         "client_id": CLIENT_ID,
26         "client_secret": CLIENT_SECRET,
27         "redirect_uri": REDIRECT_URI,
28         "code": code,
29     }
30     headers = {"Accept": "application/json"}
31     endpoint = "https://github.com/login/oauth/access_token"
32     response = requests.post(endpoint, params=params, headers=headers).json()
33     return response["access_token"]
34
35 def print_user_info(access_token=None):
36     headers = {"Authorization": f"token {access_token}"}
37     endpoint = "https://api.github.com/user"
38     response = requests.get(endpoint, headers=headers).json()
39     name = response["name"]
40     username = response["login"]
41     private_repos_count = response["total_private_repos"]
42     print(
43         f"{name} ({username}) | private repositories: {private_repos_count}"
44     )
45
46 link = create_oauth_link()
47 print(f"Follow the link to start the authentication with GitHub: {link}")
48 code = input("GitHub code: ")
49 access_token = exchange_code_for_access_token(code)
50 print(f"Exchanged code {code} with access token: {access_token}")
51 print_user_info(access_token=access_token)
```

Here's what happens when you run the code above:

1. A link is generated asking you to go to a GitHub page for authentication.
2. After following that link and logging in with your GitHub credentials, you're redirected to your defined callback URL with a code field in the query parameters:

```
{
  "args": {
    "code": "4897163087a4b153b224"
  },
  "data": "",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image",
    "Accept-Encoding": "gzip, deflate, br",
    "Accept-Language": "en-GB,en;q=0.9",
    "Host": "httpbin.org",
    "Referer": "https://github.com/",
    "Sec-Fetch-Dest": "document",
    "Sec-Fetch-Mode": "navigate",
    "Sec-Fetch-Site": "cross-site",
    "Sec-Fetch-User": "?1",
    "Upgrade-Insecure-Requests": "1",
    "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_6) AppleWebKit/537.36",
    "X-Amzn-Trace-Id": "Root=1-5fc2a19a-6aa207db14cafd4043272be"
  },
  "json": null,
  "method": "GET",
  "origin": "80.216.7.24",
  "url": "https://httpbin.org/anything?code=4897163087a4b153b224"
}
```

Example GitHub OAuth Code

3. After pasting that code in your console, you exchange the code for a reusable access token.
4. Your user information is fetched using that access token. Your name, username, and private repositories count are printed.

If you follow the steps above, then you should get a similar end result to this one:

Shell

```
$ John Doe (johndoe) | number of private repositories: 42
```

There are quite a few steps to take here, but it's important that you take the time to really understand each one. Most APIs using OAuth will share a lot of the same behavior, so knowing this process well will unlock a lot of potential when you're reading data from APIs.

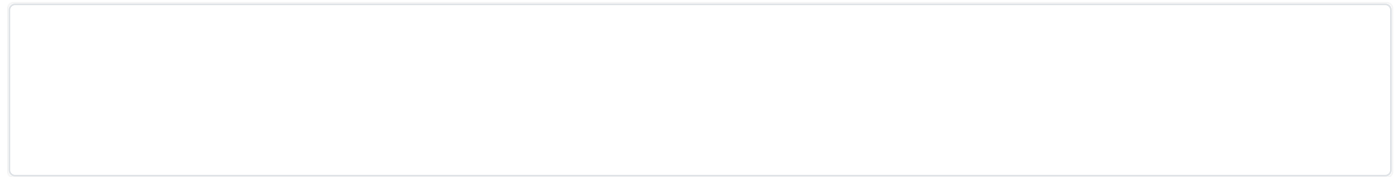
Feel free to improve this example and add more functionality, such as getting your public and starred repositories or iterating through your followers to identify the most popular ones.

There are plenty of great resources online about OAuth, and if consuming APIs behind OAuth is what you really need, then I'd advise you to do a bit more research on that topic specifically. Here are a few good places to start:

- [What the Heck is OAuth?](#)

- [OAuth 2 Simplified](#)
- [OAuth 2.0 Authorization Framework](#)

From an API consumption perspective, knowing OAuth will definitely come very in handy when you're interacting with public APIs. Most APIs have adopted OAuth as their authentication standard, and with good reason.

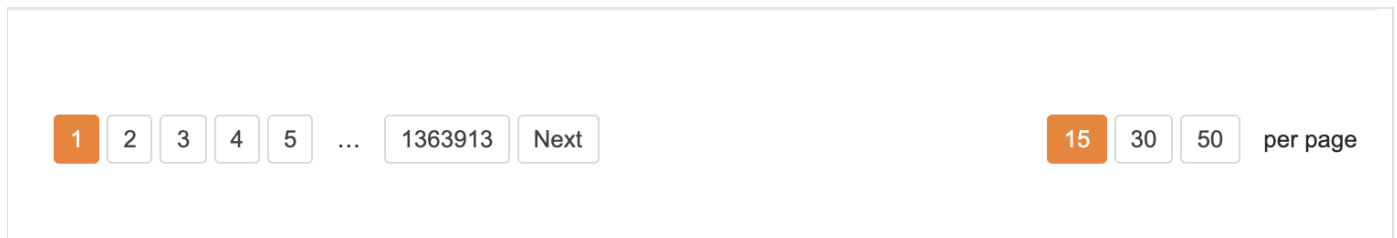


Remove ads

Pagination

Sending lots of data back and forth between clients and servers comes with a price: bandwidth. To make sure that servers can cope with a lot of requests, APIs typically use **pagination**.

In very simple terms, pagination is the act of splitting large amounts of data into multiple smaller pieces. For example, whenever you go to the [questions page](#) in Stack Overflow, you see something like this at the bottom:



Example Pagination in Stack Overflow

You probably recognize this from many other websites, and the concept is mostly the same across different sites. For APIs in specific, this is normally handled with the help of query parameters, mainly the following two:

1. A page attribute that defines which page you're currently requesting
2. A size attribute that defines the size of each page

The specific query parameter names might vary a lot depending on the API developers, but the concept is the same. A few API players might also use HTTP headers or the JSON response to return current pagination filters in place.

Using the GitHub API again, you can find an [events](#) endpoint in the documentation that contains pagination query parameters. The parameter `per_page=` defines the number of items to return, and `page=` allows you to paginate through multiple results. Here's how to use these parameters:

Python

>>>

```
>>> response = requests.get("https://api.github.com/events?per_page=1&page=0")
>>> response.json()[0]["id"]
'14345572615'
>>> response = requests.get("https://api.github.com/events?per_page=1&page=1")
>>> response.json()[0]["id"]
'14345572808'
>>> response = requests.get("https://api.github.com/events?per_page=1&page=2")
>>> response.json()[0]["id"]
'14345572100'
```

With the [first URL](#), you're only able to fetch one event. But using the `page=` query parameter, you can keep paginating through results, making sure that you're able to fetch all of the events without overloading the API.

Rate Limiting

Given that APIs are public facing and can be used by anyone, people with bad intentions often try to abuse them. To prevent such attacks, you can use a technique called **rate limiting**, which restricts the number of requests that users can make in a given time frame.

Some APIs may actually block your IP or API keys if you go over the defined rate limit too often. Be careful not to exceed the limits set by the API developers. Otherwise, you might have to wait a while before calling that API again.

For the example below, you'll once again use the GitHub API and the `/events` endpoint. According to its [documentation](#), GitHub allows about sixty unauthenticated requests per hour. If you go above that, then you'll get a 403 status code and won't be able to make any more API calls for quite some time.

Warning: Running the next piece of code will really block you from calling GitHub for some time, so make sure you don't need access to GitHub's API for a bit before you run it.

For the sake of demonstration, you'll purposefully try to exceed GitHub's rate limit to see what happens. In the code below, you'll request data until you get a status code other than 200 OK:

Python

>>>

```
>>> endpoint = "https://api.github.com/events"
>>> for i in range(100):
>>>     response = requests.get(endpoint)
>>>     print(f"{i} - {response.status_code}")
>>>     if response.status_code != 200:
>>>         break
0 - 200
1 - 200
2 - 200
3 - 200
4 - 200
5 - 200
...
55 - 200
56 - 200
57 - 403
>>> response
<Response [403]>
>>> response.json()
{'message': 'API rate limit exceeded for <ip-address>.',
 'documentation_url': 'https://developer.github.com/v3/#rate-limiting'}
```

There you have it: After about sixty requests, the API stopped returning 200 OK responses and returned a 403 Forbidden response instead, informing you that you exceeded the API rate limit.

Some APIs, like GitHub's, might even include additional information in the headers regarding your current rate limit and how many requests you have remaining. These are very helpful for you to avoid going over the defined limit. Have a look at the latest `response.headers` to see if you can find those specific rate limiting headers.

Consuming APIs With Python: Practical Examples

Now that you know all the theory and have experimented with a few APIs, you can consolidate those learnings with some more practical examples. You can modify the examples below to tailor them to your own purposes.

You can follow along with the examples by downloading the source code available at the link below:

Get the Source Code: [Click here to get the source code you'll use](#) to learn about consuming APIs with Python in this tutorial.

 Remove ads

Searching and Fetching Trending GIFs

How about making a small script that fetches the top three trending GIFs from the [GIPHY](#) website? To do this, you need to create an app and get an API key from GIPHY. You can find instructions by expanding the box below, and you can also check GIPHY's [quickstart documentation](#).

Creating a GIPHY App

Show/Hide

After you have the API key in your hands, you can start writing some code to consume this API. However, sometimes you want to run some tests before you implement a lot of code. I know I do. The thing is, a few APIs will actually provide you with tools to fetch API data directly from the documentation or their dashboard.

In this particular case, GIPHY provides you with an [API Explorer](#) that, after you create your app, allows you to start consuming the API without writing a line of code.

Some other APIs will provide you with explorers within the documentation itself, which is what [TheDogAPI](#) does on the bottom of each API reference page.

In any case, you can always use code to consume APIs, and that's what you'll do here. Grab the API key from the dashboard and, by replacing the value of the API_KEY variable below, you can start consuming the GIPHY API:

Python

```
1 import requests
2
3 # Replace the following with the API key generated.
4 API_KEY = "API_KEY"
5 endpoint = "https://api.giphy.com/v1/gifs/trending"
6
7 params = {"api_key": API_KEY, "limit": 3, "rating": "g"}
8 response = requests.get(ENDPOINT, params=params).json()
9 for gif in response["data"]:
10     title = gif["title"]
11     trending_date = gif["trending_datetime"]
12     url = gif["url"]
13     print(f"{title} | {trending_date} | {url}")
```

At the top of the file, on lines 4 and 5, you define your `API_KEY` and the GIPHY API endpoint since they won't change as often as the rest.

On line 7, making use of what you learned in the [query parameters](#) section, you define the params and add your own API key. You also include a couple of other filters: limit to get 3 results and rating to get only [appropriate content](#).

Finally, after getting a response, you [iterate](#) through the results on line 9. For each GIF, you print its title, date, and URL on line 13.

Running this piece of code in the console would output a somewhat structured list of GIFs:

Shell

```
Excited Schitts Creek GIF by CBC | 2020-11-28 20:45:14 | https://giphy.com/gifs/cbc-schittscreek-schitts-creek-SiGg4zSmwmbafTYwpj
Saved By The Bell Shrug GIF by PeacockTV | 2020-11-28 20:30:15 | https://giphy.com/gifs/peacocktv-saved-by-the-bell-bayside-high-sch
Schitts Creek Thank You GIF by CBC | 2020-11-28 20:15:07 | https://giphy.com/gifs/cbc-funny-comedy-26n79l9afmfm1POjC
```

Now, let's say you want to make a script that allows you to search for a specific word and fetch the first GIPHY match to that word. A different endpoint and slight variation of the code above can do that quite quickly:

Python

```
import requests

# Replace the following with the API key generated.
API_KEY = "API_KEY"
endpoint = "https://api.giphy.com/v1/gifs/search"

search_term = "shrug"
params = {"api_key": API_KEY, "limit": 1, "q": search_term, "rating": "g"}
response = requests.get(endpoint, params=params).json()
for gif in response["data"]:
    title = gif["title"]
    url = gif["url"]
    print(f"{title} | {url}")
```

There you have it! Now you can modify this script to your liking and generate GIFs on demand. Try fetching GIFs from your favorite show or movie, adding a shortcut to your terminal to get the most popular GIFs on demand, or integrating with another API from your favorite messaging system—WhatsApp, Slack, you name it. Then start sending GIFs to your friends and coworkers!

Getting COVID-19 Confirmed Cases Per Country

Even though this may be something that you're tired of hearing about by now, there's a [free API](#) with up-to-date world COVID-19 data. This API doesn't require authentication, so it's pretty straightforward to get some data right away. The free version that you'll use below has a rate limit and some restrictions on the data, but it's more than enough for small use cases.

For this example, you'll get the total number of confirmed cases up to the previous day. I randomly picked Germany again as the country, but you can pick any [country slug](#) you like:

Python

```
1 import requests
2 from datetime import date, timedelta
3
4 today = date.today()
5 yesterday = today - timedelta(days=1)
6 country = "germany"
7 endpoint = f"https://api.covid19api.com/country/{country}/status/confirmed"
8 params = {"from": str(yesterday), "to": str(today)}
9
10 response = requests.get(endpoint, params=params).json()
11 total_confirmed = 0
12 for day in response:
13     cases = day.get("Cases", 0)
14     total_confirmed += cases
15
16 print(f"Total Confirmed Covid-19 cases in {country}: {total_confirmed}")
```

On lines 1 and 2, you import the necessary modules. In this case, you have to import the `date` and `timedelta` objects to be able to get today's and yesterday's dates.

On lines 6 to 8, you define the country slug you want to use, the endpoint, and the query parameters for the API request.

The response is a list of days, and for each day you have a `Cases` field that contains the total number of confirmed cases on that date. On line 11, you create a variable to keep the total number of confirmed cases, and then on line 14 you iterate through all the days and sum them up.

Printing the end result will show you the total number of confirmed cases in the selected country:

Shell

```
Total Confirmed Covid-19 cases in germany: 1038649
```

In this example, you're looking at total number of confirmed cases for a whole country. However, you could also try looking at the [documentation](#) and fetching the data for your specific city instead. And why not make it a bit more thorough and get some other data, such as the number of recovered cases?

 Remove ads

Searching Google Books

If you have a passion for books, then you might want a quick way to search for a specific book. You might even want to connect it to your local library's search to see if a given book is available using the book's [ISBN](#).

For this example, you'll use the [Google Books API](#) and the public [volumes endpoint](#) to do simple searches of books.

Here's a straightforward piece of code to look for the words `moby dick` in the whole catalog:

Python

```
1 import requests
2
3 endpoint = "https://www.googleapis.com/books/v1/volumes"
4 query = "moby dick"
5
6 params = {"q": query, "maxResults": 3}
7 response = requests.get(endpoint, params=params).json()
8 for book in response["items"]:
9     volume = book["volumeInfo"]
10    title = volume["title"]
11    published = volume["publishedDate"]
12    description = volume["description"]
13    print(f"{title} ({published}) | {description}")
```

This code example is pretty similar to the ones you've seen before. You start on lines 3 and 4 by defining important variables, such as the endpoint and, in this case, the query.

After making the API request, on line 8 you start iterating through the results. Then, on line 13, you print the most interesting information for each book that matches your initial query:

Shell

```
Moby-Dick (2016-04-12) | "Call me Ishmael." So begins the famous opening...
Moby Dick (1892) | A literary classic that wasn't recognized for its...
Moby Dick; Or, The Whale (1983-08-16) | The story of Captain Ahab's...
```

You can print the book variable inside the loop to see what other fields you have available. Here are a few that could be useful for further improving this code:

- industryIdentifiers
- averageRating and ratingsCount
- imageLinks

A fun challenge to do with this API is to use your [OAuth knowledge](#) and create your own bookshelf app that keeps records of all the books you read or want to read. You can even connect it to your favorite bookstore or library afterward to quickly find books from your wish list that are available near you. This is just one idea—I'm sure you can come up with more.

Conclusion

There are a million other things you can learn about APIs: different headers, different content types, different authentication techniques, and so on. However, the concepts and techniques you learned in this tutorial will allow you to practice with any API of your liking and to use Python for any API consumption needs you may have.

In this tutorial, you learned:

- What an **API** is and what can you use it for
- What **status codes**, **HTTP headers**, and **HTTP methods** are
- How can you use Python to **consume public data** using APIs
- How to use **authentication** when consuming APIs with Python

Go ahead and try this new magic skill with some public APIs of your liking! You can also review the examples you

saw in this tutorial by downloading the source code from the link below:

Get the Source Code: [Click here to get the source code you'll use](#) to learn about consuming APIs with Python in this tutorial.

Further Reading

The APIs used as examples in this tutorial are just a tiny fraction of the numerous public APIs available for free. Here's a list of API collections that you can use to find your next favorite API:

- [GitHub Public APIs repository](#)
- [Public APIs](#)
- [Public API](#)
- [Any API](#)

You can check these out and find an API that speaks to you and your hobbies and maybe inspires you to do a small project with it. If you come across a good public API that you think I or other people reading the tutorial should know about, then please leave a comment below!

Mark as Completed



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}
```

Email Address

About **Pedro Pregueiro**



Hi! My name is Pedro and I'm a Python developer who loves coding, burgers and playing guitar.

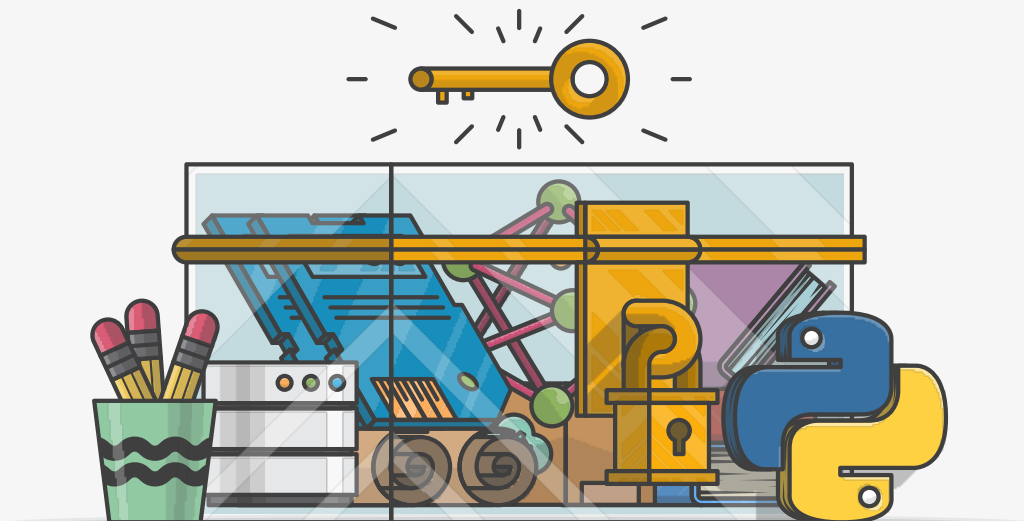
[» More about Pedro](#)

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:





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



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

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.

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.

Keep Learning

Related Tutorial Categories: [api](#) [intermediate](#)

— 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.

All Tutorial Topics

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

Table of Contents

- [Getting to Know APIs](#)
- [Calling Your First API Using Python](#)
- [Learning Advanced API Concepts](#)
- [Consuming APIs With Python: Practical Examples](#)
- [Conclusion](#)
- [Further Reading](#)

Mark as Completed



 [Tweet](#)

 [Share](#)

 [Email](#)

© 2012–2021 Real Python · [Newsletter](#) · [Podcast](#) · [YouTube](#) · [Twitter](#) · [Facebook](#) ·
[Instagram](#) · [Python Tutorials](#) · [Search](#) · [Privacy Policy](#) · [Energy Policy](#) · [Advertise](#) ·
[Contact](#)

♥ Happy Pythoning!