CONTENTS

RELATED

How To Setup uWSGI On Ubuntu 12.10
View

How To Create Nagios Plugins With Python On CentOS 6
View

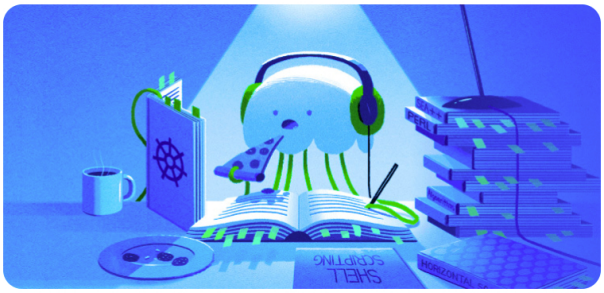
// Tutorial //

Getting Started With Python Requests - GET Requests

Published on September 21, 2020 · Updated on September 14, 2020

Python

By Anthony Herbert



While we believe that this content benefits our community, we have not yet thoroughly reviewed it. If you have any suggestions for improvements, please let us know by clicking the "report an issue" button at the bottom of the tutorial.

Introduction

In many web apps, it's normal to connect to various third-party services by using APIs. When you use these APIs you can get access to data like weather information, sports scores, movie listings, tweets, search engine results, and pictures. You can also use APIs to add functionality to your app. Examples of these are payments, scheduling, emails, translations, maps, and file transfers. If you were to create any of those on your own it would take a ton of time, but with APIs, it can take only minutes to connect to one and access its features and data.

In this article, we'll learn about the Python Requests library, which allows you to send HTTP requests in Python.

And since using an API is sending HTTP requests and receiving responses, Requests allows you to use APIs in Python. We'll demonstrate the use of a language translation API here so you can see an example of how it works.

Quick Overview of HTTP Requests

HTTP requests are how the web works. Every time you navigate to a web page, your browser makes multiple requests to the web page's server. The server then responds with all the data necessary to render the page, and your browser then actually renders the page so you can see it.

The generic process is this: a client (like a browser or Python script using Requests) will send some data to a URL, and then the server located at the URL will read the data, decide what to do with it, and return a response to the client. Finally, the client can decide what to do with the data in the response.

Part of the data the client sends in a request is the request method. Some common request methods are GET, POST, and PUT. GET requests are normally for reading data only without making a change to something, while POST and PUT requests generally are for modifying data on the server. So for example, the Stripe API allows you to use POST requests to create a new charge so a user can purchase something from your app.

Note: This article will cover GET requests, because we won't be modifying any data on a server.

When sending a request from a Python script or inside a web app, you, the developer, gets to decide what gets sent in each request and what to do with the response. So let's explore that by first sending a request to Scotch.io and then by using a language translation API.

Install Python Requests

Before we can do anything, we need to install the library. So let's go ahead and install requests using `pip`. It's a good idea to create a virtual environment first if you don't already have one.

```
$ pip install requests
```

Copy

Our First Request

To start, let's use Requests for requesting the Scotch.io site. Create a file called `script.py` and add the following code to it. In this article, we won't have much code to work with, so when something changes you can just update the existing code instead of adding new lines.

script.py

```
import requests

res = requests.get('https://scotch.io')

print(res)
```

Copy

Popular Topics

Ubuntu
Linux Basics
JavaScript
Python
MySQL
Docker
Kubernetes

All tutorials →

Free Managed Hosting →

So all this code is doing is sending a GET request to Scotch.io. This is the same type of request your browser sent to view this page, but the only difference is that Requests can't actually render the HTML, so instead you will just get the raw HTML and the other response information.

We're using the `.get()` function here, but Requests allows you to use other functions like `.post()` and `.put()` to send those requests as well.

You can run it by executing the script.py file.

```
$ python script.py
```
Copy

And here's what you get in return:

Ubuntu
```
> python script.py
<Response [200]>
>
```

Status Codes

The first thing we can do is check the status code. HTTP codes range from the 1XX to 5XX. Common status codes that you have probably seen are 200, 404, and 500.

Here's a quick overview of what each status code means:

- 1XX - Information
- 2XX - Success
- 3XX - Redirect
- 4XX - Client Error (you made an error)
- 5XX - Server Error (they made an error)

Generally, what you're looking for when you perform your own requests are status codes in the 200s.

Requests recognizes that 4XX and 5XX status codes are errors, so if those status codes get returned, the response object from the request evaluates to `False`.

You can test if a request responded successfully by checking the response for truth. For example:

script.py
```
if res:
    print('Response OK')
else:
    print('Response Failed')
```
Copy

Ubuntu
```
> python script.py
<Response [200]>
Response OK
>
```

The message "Response Failed" will only appear if a 400 or 500 status code returns. Try changing the URL to some nonsense to see the response fail with a 404.

You can take a look at the status code directly by adding:

script.py
```
print(res.status_code)
```
Copy

This will show you the status code directly so you can check the number yourself.

Ubuntu
```
> python script.py
<Response [404]>
Response Failed
404
>
```

Headers

Another thing you can get from the response are the headers. You can take a look at them by using the headers dictionary on the response object.

script.py
```
print(res.headers)
```
Copy

Headers are sent along with the request and returned in the response. Headers are used so both the client and the server know how to interpret the data that is being sent and received in the response/response.

We see the various headers that are returned. A lot of times you won't need to use the header information directly, but it's there if you need it.

The content type is usually the one you may need because it reveals the format of the data, for example HTML, JSON, PDF, text, etc. But the content type is normally handled by Requests so you can access the data that gets returned.

Response Text

And finally, if we take a look at `res.text` (this works for textual data, like a HTML page like we are viewing) we can see all the HTML needed to build the home page of Scotch. It won't be rendered, but we see that it looks like it belongs to Scotch. If you saved this to a file and opened it, you would see something that resembled the Scotch site. In a real situation, multiple requests are made for a single web page to load things like images, scripts, and stylesheets, so if you save only the HTML to a file, it won't look anything like what the Scotch.io page looks like in your browser because only a single request was performed to get the HTML data.

script.py

```
print(res.text)
```

Copy

Using the Translate API

So now let's move on to something more interesting. We'll use the Yandex Translate API to perform a request to translate some text to a different language.

To use the API, first you need to sign up. After you sign up, go to the Translate API and create an API key. Once you have the API key, add it to your file as a constant. Here's the link where you can do all those things: https://tech.yandex.com/translate/

script.py

```
API_KEY = 'your yandex api key'
```

Copy

The reason why we need an API key is so Yandex can authenticate us every time we want to use their API. The API key is a lightweight form of authentication, because it's added on to the end of the request URL when being sent.

To know which URL we need to send to use the API, we can look at the documentation for Yandex.

If we look there, we'll see all the information needed to use their Translate API to translate text.

Request syntax

```
https://translate.yandex.net/api/v1.5/tr.json/translate
 ? key=<API key>
 & text=<text to translate>
 & lang=<translation direction>
 & [format=<text format>]
 & [options=<translation options>]
 & [callback=<name of the callback function>]
```

When we see a URL with ampersands (&), question marks (?), and equals signs (=), you can be sure that the URL is for GET requests. Those symbols specify the parameters that go along with the URL.

Normally things in square brackets ([]) will be optional. In this case, format, options, and callback are optional, while the key, text, and lang are required for the request.

So let's add some code to send to that URL. You can replace the first request we created with this:

script.py

```
url = 'https://translate.yandex.net/api/v1.5/tr.json/translate'
res = requests.get(url)
```

Copy

There are two ways we can add the parameters. We can either append it to the end of the URL directly, or we can have Requests do it for us. To do the latter, we can create a dictionary for our parameters. The three items we need are the key, the text, and the language. Let's create the dictionary using the API key, `'Hello'` for the text, and `'en-es'` as the lang, which means we want to translate from English to Spanish.

If you need to know any other language codes, you can look here. You are looking for the 639-1 column.

We create a params dictionary by using the `dict()` function and passing in the keys and values we want in our dictionary.

script.py

```
params = dict(key=API_KEY, text='Hello', lang='en-es')
```

Copy

Now we take the parameters dictionary and pass it to the `.get()` function.

script.py

```
res = requests.get(url, params=params)
```

Copy

When we pass the parameters this way, Requests will go ahead and add the parameters to the URL

When we pass the parameters this way, Requests will go ahead and add the parameters to the URL for us.

Now let's add a print statement for the response text and view what gets returned in the response.

script.py

```
print(res.text)
```

Copy

Ubuntu

```
> python script.py
{"code":200,"lang":"en-es","text":["Hola"]}
>
```

We see three things. We see the status code, which is exactly the same status code of the response itself, we see the language that we specified, and we see the translated text inside of the list. So you should see `'Hola'` for the translated text.

Try again with en-fr as the language code, and you should see `'Bonjour'` in the response now.

script.py

```
params = dict(key=API_KEY, text='Hello', lang='en-fr')
```

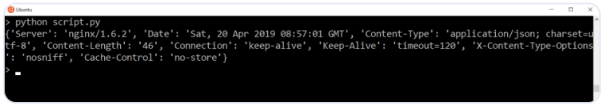
Copy

Let's take a look at the headers for this particular response.

script.py

```
print(res.headers)
```

Copy



Obviously the headers should be different because we're communicating with a different server, but in this case the content type is application/json instead of text/html. What this means that the data can be interpreted as JSON.

When application/json is the content type of the response, we are able to have Requests convert the response to a dictionary and list so we can access the data easier.

To have the data parsed as JSON, we use the `.json()` method on the response object.

If you print it, you'll see that the data looks the same, but the format is slightly different.

script.py

```
json = res.json()
print(json)
```

Copy

Ubuntu

```
> python script.py
{'code': 200, 'lang': 'en-fr', 'text': ['Bonjour']}
>
```

The reason why it's different is because it's no longer plain text that you get from res.text. This time it's a printed version of a dictionary.

Let's say we want to access the text. Since this is now a dictionary, we can use the text key.

script.py

```
print(json['text'])
```

Copy

Ubuntu

```
> python script.py
['Bonjour']
>
```

And now we only see the data for that one key. In this case we are looking at a list of one item, so if we wanted to get that text in the list directly, we can access it by the index.

script.py

```
print(json['text'][0])
```

Copy

And now the only thing we see is the translated word.

So of course if we change things in our parameters, we'll get different results. Let's change the text to be translated from `Hello` to `Goodbye`, change the target language back to Spanish, and send the request again.

script.py

```
params = dict(key=API_KEY, text='Goodbye', lang='en-es')
```

Copy

Ubuntu

```
> python script.py
```

```
Adiós
>
```

Try translating longer text in different languages and see what responses the API gives you.

Translate API Error Cases

Finally, we'll take a look at an error case. Everything doesn't always work, so we need to know when that happens.

Try changing your API key by removing one character. When you do this your API key will no longer be valid. Then try sending a request.

If you take a look at the status code, this is what you get:

script.py

```
print(res.status_code)
```

Copy

Ubuntu

```
> python script.py
403
>
```

So when you are using the API, you'll want to check if things are successful or not so you can handle the error cases according to the needs of your app.

Conclusion

Here's what we learned:

- How HTTP requests work
- The various status codes possible in a response
- How to send requests and receive responses using the Python Requests library
- How to use a language translation API to translate text
- How to convert application/JSON content responses to dictionaries

If you want to do more, check out this list to see different APIs that are available, and try to use them with Python Requests.

About the authors

Anthony Herbert Author

marketplace offerings and insert the link!

Sign In or Sign Up to Comment

Get our biweekly newsletter

Sign up for Infrastructure as a Newsletter.

Sign up →

Hollie's Hub for Good

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to help.

Learn more →

Become a contributor

You get paid; we donate to tech nonprofits.

Learn more →

Featured on Community

Kubernetes Course · Learn Python 3 · Machine Learning in Python · Getting started with Go · Intro to Kubernetes

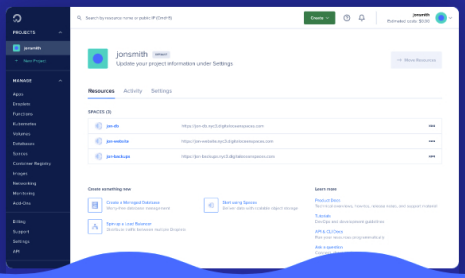
DigitalOcean Products

Cloudways · Virtual Machines · Managed Databases · Managed Kubernetes · Block Storage · Object Storage · Marketplace · VPC · Load Balancers

Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow – whether you're running one virtual machine or ten thousand.

Learn more →



Company

About
Leadership
Blog
Careers
Customers
Partners
Channel Partners
Referral Program
Affiliate Program
Press
Legal
Security
Investor Relations
DO Impact

Products

Products Overview
Droplets
Kubernetes
App Platform
Functions
Cloudways
Managed Databases
Spaces
Marketplace
Load Balancers
Block Storage
Tools & Integrations
API
Pricing
Documentation
Release Notes
Uptime

Community

Tutorials
Q&A
CSS-Tricks
Write for DOnations
Currents Research
Hatch Startup Program
deploy by DigitalOcean
Shop Swag
Research Program
Open Source
Code of Conduct
Newsletter Signup
Meetups

Solutions

Website Hosting
VPS Hosting
Web & Mobile Apps
Game Development
Streaming
VPN
SaaS Platforms
Cloud Hosting for Blockchain
Startup Resources

Contact

Support
Sales
Report Abuse
System Status
Share your ideas