tuts+

CODE > PYTHON

Using the Requests Module in Python

by Monty Shokeen 9 Mar 2017

Difficulty: Intermediate Length: Medium Languages: English

Python Web Development

Requests is a Python module that you can use to send all kinds of HTTP requests. It is an easy-to-use library with a lot of features ranging from passing parameters in URLs to sending custom headers and SSL Verification. In this tutorial, you will learn how to use this library to send simple HTTP requests in Python.

You can use Requests with Python version 2.6–2.7 and 3.3–3.6. Before proceeding further, you should know that Requests is an external module, so you will have to install it first before trying out the examples in this tutorial. You can install it by running the following command in the terminal:

```
pip install requests
```

Once you have installed the module, you can verify if it has been successfully installed by importing it using this command:

```python
import requests
```

If the installation has been successful, you won't see any error messages.

Making a GET Request

It is very easy to send an HTTP request using Requests. You begin by importing the module and then make the request. Here is an example:

```
import requests
req = requests.get('https://tutsplus.com/')
```

All the information about our request is now stored in a Response object called `req`. For example, you can get the encoding of the webpage using the `req.encoding` property. You can also get the status code of the request using the `req.status_code` property.

```
req.encoding     # returns 'utf-8'
req.status_code  # returns 200
```

You can access the cookies that the server sent back using `req.cookies`. Similarly, you can get the response headers using `req.headers`. The `req.headers` property returns a case insensitive dictionary of response headers. This means that `req.headers['Content-Length']`, `req.headers['content-length']` and `req.headers['CONTENT-LENGTH']` will all return the value of the `'Content-Length'` response header.

You can check if the response is a well-formed HTTP redirect that could have been processed automatically using the `req.is_redirect` property. It will return `True` or `False` based on the response. You can also get the time elapsed between sending the request and getting back a response using the `req.elapsed` property.

The URL that you initially passed to the `get()` function can be different than the final URL of the response for a variety of reasons, including redirects. To see the final response URL, you can use the `req.url` property.

```
import requests
req = requests.get('http://www.tutsplus.com/')

req.encoding      # returns 'utf-8'
req.status_code   # returns 200
req.elapsed       # returns datetime.timedelta(0, 1, 666890)
req.url           # returns 'https://tutsplus.com/'

req.history
# returns [<Response [301]>, <Response [301]>]
```

```
req.headers['Content-Type']
# returns 'text/html; charset=utf-8'
```

Getting all this information about the webpage you are accessing is nice, but you most probably want to access the actual content. If the content you are accessing is text, you can use the `req.text` property to access it. The content is then parsed as unicode. You can pass the encoding with which to decode the text using the `req.encoding` property.

In the case of non-text responses, you can access them in binary form using `req.content`. The module will automatically decode `gzip` and `deflate` transfer-encodings. This can be helpful when you are dealing with media files. Similarly, you can access the json-encoded content of the response, if it exists, using `req.json()`.

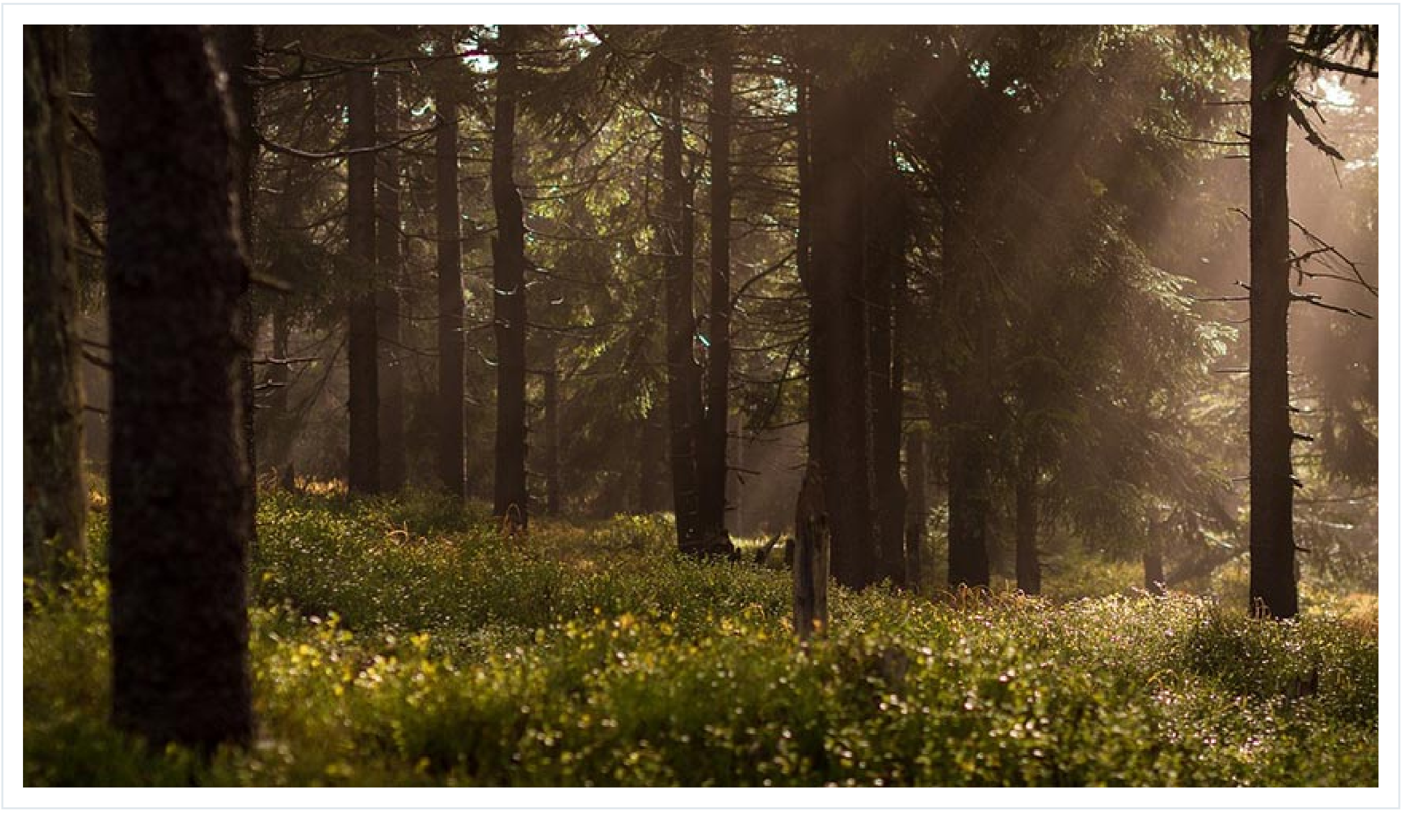
You can also get the raw response from the server using `req.raw`. Keep in mind that you will have to pass `stream=True` in the request to get the raw response.

Some files that you download from the internet using the Requests module may have a huge size. In such cases, it will not be wise to load the whole response or file in the memory at once. You can download a file in pieces or chunks using the `iter_content(chunk_size = 1, decode_unicode=False)` method.

This method iterates over the response data in `chunk_size` number of bytes at once. When `stream=True` has been set on the request, this method will avoid reading the whole file into memory at once for large responses. The `chunk_size` parameter can be either an integer or `None`. When set to an integer value, `chunk_size` determines the number of bytes that should be read into the memory.

When `chunk_size` is set to `None` and `stream` is set to `True`, the data will be read as it arrives in whatever size of chunks are received. When `chunk_size` is set to `None` and `stream` is set to `False`, all the data will be returned as a single chunk.

Let's download this image of a forest on Pixabay using the Requests module. Here is the actual image:



This is the code that you need:

```
import requests
req = requests.get('path/to/forest.jpg', stream=True)
req.raise_for_status()
with open('Forest.jpg', 'wb') as fd:
    for chunk in req.iter_content(chunk_size=50000):
        print('Received a Chunk')
        fd.write(chunk)
```

The `'path/to/forest.jpg'` is the actual image URL; you can put the URL of any other image here to download something else. The given image file is 185kb in size, and you have set `chunk_size` to 50,000 bytes. This means that the "Received a Chunk" message should be printed four times in the terminal. The size of the last chunk will just be 39350 bytes because the part of the file that remains to be received after the first three iterations is 39350 bytes.

Requests also allows you to pass parameters in a URL. This can be helpful when you are searching a webpage for some results like a specific image or tutorial. You can provide these query strings as a dictionary of strings using the `params` keyword in the GET request. Here is an example:

```
import requests

query = {'q': 'Forest', 'order': 'popular', 'min_width': '800', 'min_height':
```

```
req = requests.get('https://pixabay.com/en/photos/', params=query)

req.url
# returns 'https://pixabay.com/en/photos/?order=popular&min_height=600&q=Fore
```

Making a POST Request

Making a POST request is just as easy as making GET requests. You just use the `post()` function instead of `get()`. This can be useful when you are automatically submitting forms. For example, the following code will download the whole Wikipedia page on Nanotechnology and save it on your PC.

```
import requests
req = requests.post('https://en.wikipedia.org/w/index.php', data = {'search':
req.raise_for_status()
with open('Nanotechnology.html', 'wb') as fd:
    for chunk in req.iter_content(chunk_size=50000):
        fd.write(chunk)
```

Sending Cookies and Headers

As previously mentioned, you can access the cookies and headers that the server sends back to you using `req.cookies` and `req.headers`. Requests also allows you to send your own custom cookies and headers with a request. This can be helpful when you want to, let's say, set a custom user agent for your request.

To add HTTP headers to a request, you can simply pass them in a `dict` to the `headers` parameter. Similarly, you can also send your own cookies to a server using a `dict` passed to the `cookies` parameter.

```
import requests

url = 'http://some-domain.com/set/cookies/headers'

headers = {'user-agent': 'your-own-user-agent/0.0.1'}
cookies = {'visit-month': 'February'}

req = requests.get(url, headers=headers, cookies=cookies)
```

Cookies can also be passed in a Cookie Jar. They provide a more complete interface to allow you to use those cookies over multiple paths. Here is an example:

```python
import requests

jar = requests.cookies.RequestsCookieJar()
jar.set('first_cookie', 'first', domain='httpbin.org', path='/cookies')
jar.set('second_cookie', 'second', domain='httpbin.org', path='/extra')
jar.set('third_cookie', 'third', domain='httpbin.org', path='/cookies')

url = 'http://httpbin.org/cookies'
req = requests.get(url, cookies=jar)

req.text

# returns '{ "cookies": { "first_cookie": "first", "third_cookie": "third" }
```

Session Objects

Sometimes it is useful to preserve certain parameters across multiple requests. The Session object does exactly that. For example, it will persist cookie data across all requests made using the same session. The Session object uses urllib3's connection pooling. This means that the underlying TCP connection will be reused for all the requests made to the same host. This can significantly boost the performance. You can also use methods of the Requests object with the Session object.

Here is an example of multiple requests sent with and without using sessions:

```python
import requests

reqOne = requests.get('https://tutsplus.com/')
reqOne.cookies['_tuts_session']
#returns 'cc118d94a84f0ea37c64f14dd868a175'

reqTwo = requests.get('https://code.tutsplus.com/tutorials')
reqTwo.cookies['_tuts_session']
#returns '3775e1f1d7f3448e25881dfc35b8a69a'

ssnOne = requests.Session()
ssnOne.get('https://tutsplus.com/')
ssnOne.cookies['_tuts_session']
#returns '4c3dd2f41d2362108fbb191448eab3b4'

reqThree = ssnOne.get('https://code.tutsplus.com/tutorials')
reqThree.cookies['_tuts_session']
#returns '4c3dd2f41d2362108fbb191448eab3b4'
```

As you can see, the session cookie has a different value in the first and second request, but it has the same value when we used the Session object. You will be getting a different value when you try out this code, but in your case too, the cookie for the requests made using the session object will have the same value.

Sessions are also helpful when you want to send the same data across all requests. For example, if you decide to send a cookie or a user-agent header with all the requests to a given domain, you can use Session objects. Here is an example:

```python
import requests

ssn = requests.Session()
ssn.cookies.update({'visit-month': 'February'})

reqOne = ssn.get('http://httpbin.org/cookies')
print(reqOne.text)
# prints information about "visit-month" cookie

reqTwo = ssn.get('http://httpbin.org/cookies', cookies={'visit-year': '2017'
print(reqTwo.text)
# prints information about "visit-month" and "visit-year" cookie

reqThree = ssn.get('http://httpbin.org/cookies')
print(reqThree.text)
# prints information about "visit-month" cookie
```

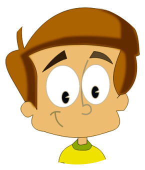
As you can see, the `"visit-month"` session cookie is sent with all three requests. However, the `"visit-year"` cookie is sent only during the second request. There is no mention of the `"vist-year"` cookie in the third request too. This confirms the fact that cookies or other data set on individual requests won't be sent with other session requests.

Conclusion

The concepts discussed in this tutorial should help you make basic requests to a server by passing specific headers, cookies, or query strings. This will be very handy when you are trying to scrape some webpages for information. Now, you should also be able to automatically download music files and wallpapers from different websites once you have figured out a pattern in the URLs.

Don’t hesitate to see what we have available for sale and for study in the marketplace, and don't hesitate to ask any questions and provide your valuable feedback using the feed below.

If you have any questions regarding this tutorial, please let me know in the comments.



Monty Shokeen

I am a full-stack developer who also loves to write tutorials in his free time. Other than that, I love learning about new and interesting JavaScript libraries.

Email Address

Update me weekly

Learn the basics of PHP testing.
Easy to follow lessons and premium video content.
FREE COURSE

Advertisement

3 Comments **Tuts+ Hub** 1 **Login**

Recommend 41 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Punith • 16 days ago

Hi,
Thanks for the explanation
Im unable to handle redirection in the below link
"https://t.co/0vDGcm1oqh"

Can you please check

• Reply • Share ›

my_junk • 6 months ago

GREAT intro to "Requests" - better than any I have seen so far. I'm quite new. I'm trying to do a simple(?) task ... "page scrape" usage daily from my DSL modem to keep under the ISP data cap. I have to remotely log into the modem:
requests.get('<modem url="">', auth=('webtest', 'modem-pswd'))
... but the login won't happen on a <return> keypress ... it must have an "Apply" button 'clicked'. I haven't figured out how to do that. I can, however, fetch the data I need that loads the "DSL Status" page AFTER the login completes. So I think I just need the one missing 'click' part to accomplish the login.

Can I do this with "Requests"? I sure hope so!

Thanks again ... you provided a lot of rich information.
Blessings in abundance, all the best, & ENJOY!
Art in Carlisle, PA USA

• Reply • Share ›

Pierre-Henry Soria • 10 months ago

Thanks! Well explained. I'm sure this tutorial will be useful for me very soon

• Reply • Share ›