Requests Documentation

Release 2.24.0

Kenneth Reitz

Aug 27, 2020

Contents

Release v2.24.0. (*Installation*)

Requests is an elegant and simple HTTP library for Python, built for human beings.

Behold, the power of Requests:

```
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
'{"type":"User"...'
>>> r.json()
{'private_gists': 419, 'total_private_repos': 77, ...}
```

See similar code, sans Requests.

Requests allows you to send HTTP/1.1 requests extremely easily. There's no need to manually add query strings to your URLs, or to form-encode your POST data. Keep-alive and HTTP connection pooling are 100% automatic, thanks to urllib3.

CHAPTER 1

Beloved Features

Requests is ready for today's web.

- Keep-Alive & Connection Pooling
- International Domains and URLs
- Sessions with Cookie Persistence
- Browser-style SSL Verification
- Automatic Content Decoding
- Basic/Digest Authentication
- Elegant Key/Value Cookies
- Automatic Decompression
- Unicode Response Bodies
- HTTP(S) Proxy Support
- Multipart File Uploads
- Streaming Downloads
- Connection Timeouts
- Chunked Requests
- `.netrc` Support

Requests officially supports Python 2.7 & 3.5+, and runs great on PyPy.

CHAPTER 2

The User Guide

This part of the documentation, which is mostly prose, begins with some background information about Requests, then focuses on step-by-step instructions for getting the most out of Requests.

2.1 Installation of Requests

This part of the documentation covers the installation of Requests. The first step to using any software package is getting it properly installed.

2.1.1 $ python -m pip install requests

To install Requests, simply run this simple command in your terminal of choice:

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

2.1.2 Get the Source Code

Requests is actively developed on GitHub, where the code is always available.

You can either clone the public repository:

```
$ git clone git://github.com/psf/requests.git
```

Or, download the tarball:

```
$ curl -OL https://github.com/psf/requests/tarball/master
# optionally, zipball is also available (for Windows users).
```

Once you have a copy of the source, you can embed it in your own Python package, or install it into your site-packages easily:

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

2.2 Quickstart

Eager to get started? This page gives a good introduction in how to get started with Requests.

First, make sure that:

- Requests is *installed*
- Requests is *up-to-date*

Let's get started with some simple examples.

2.2.1 Make a Request

Making a request with Requests is very simple.

Begin by importing the Requests module:

```
>>> import requests
```

Now, let's try to get a webpage. For this example, let's get GitHub's public timeline:

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

Now, we have a *Response* object called `r`. We can get all the information we need from this object.

Requests' simple API means that all forms of HTTP request are as obvious. For example, this is how you make an HTTP POST request:

```
>>> r = requests.post('https://httpbin.org/post', data = {'key':'value'})
```

Nice, right? What about the other HTTP request types: PUT, DELETE, HEAD and OPTIONS? These are all just as simple:

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

That's all well and good, but it's also only the start of what Requests can do.

2.2.2 Passing Parameters In URLs

You often want to send some sort of data in the URL's query string. If you were constructing the URL by hand, this data would be given as key/value pairs in the URL after a question mark, e.g. `httpbin.org/get?key=val`. Requests allows you to provide these arguments as a dictionary of strings, using the `params` keyword argument. As an example, if you wanted to pass `key1=value1` and `key2=value2` to `httpbin.org/get`, you would use the following code:

```
>>> payload = {'key1': 'value1', 'key2': 'value2'}
>>> r = requests.get('https://httpbin.org/get', params=payload)
```

You can see that the URL has been correctly encoded by printing the URL:

```
>>> print(r.url)
https://httpbin.org/get?key2=value2&key1=value1
```

Note that any dictionary key whose value is `None` will not be added to the URL's query string.

You can also pass a list of items as a value:

```
>>> payload = {'key1': 'value1', 'key2': ['value2', 'value3']}

>>> r = requests.get('https://httpbin.org/get', params=payload)
>>> print(r.url)
https://httpbin.org/get?key1=value1&key2=value2&key2=value3
```

2.2.3 Response Content

We can read the content of the server's response. Consider the GitHub timeline again:

```
>>> import requests

>>> r = requests.get('https://api.github.com/events')
>>> r.text
'[{"repository":{"open_issues":0,"url":"https://github.com/...
```

Requests will automatically decode content from the server. Most unicode charsets are seamlessly decoded.

When you make a request, Requests makes educated guesses about the encoding of the response based on the HTTP headers. The text encoding guessed by Requests is used when you access `r.text`. You can find out what encoding Requests is using, and change it, using the `r.encoding` property:

```
>>> r.encoding
'utf-8'
>>> r.encoding = 'ISO-8859-1'
```

If you change the encoding, Requests will use the new value of `r.encoding` whenever you call `r.text`. You might want to do this in any situation where you can apply special logic to work out what the encoding of the content will be. For example, HTML and XML have the ability to specify their encoding in their body. In situations like this, you should use `r.content` to find the encoding, and then set `r.encoding`. This will let you use `r.text` with the correct encoding.

Requests will also use custom encodings in the event that you need them. If you have created your own encoding and registered it with the `codecs` module, you can simply use the codec name as the value of `r.encoding` and Requests will handle the decoding for you.

2.2.4 Binary Response Content

You can also access the response body as bytes, for non-text requests:

```
>>> r.content
b'[{"repository":{"open_issues":0,"url":"https://github.com/...
```

The `gzip` and `deflate` transfer-encodings are automatically decoded for you.

For example, to create an image from binary data returned by a request, you can use the following code:

```
>>> from PIL import Image
>>> from io import BytesIO

>>> i = Image.open(BytesIO(r.content))
```

2.2.5 JSON Response Content

There's also a builtin JSON decoder, in case you're dealing with JSON data:

```
>>> import requests

>>> r = requests.get('https://api.github.com/events')
>>> r.json()
[{'repository': {'open_issues': 0, 'url': 'https://github.com/...
```

In case the JSON decoding fails, `r.json()` raises an exception. For example, if the response gets a 204 (No Content), or if the response contains invalid JSON, attempting `r.json()` raises `ValueError: No JSON object could be decoded`.

It should be noted that the success of the call to `r.json()` does **not** indicate the success of the response. Some servers may return a JSON object in a failed response (e.g. error details with HTTP 500). Such JSON will be decoded and returned. To check that a request is successful, use `r.raise_for_status()` or check `r.status_code` is what you expect.

2.2.6 Raw Response Content

In the rare case that you'd like to get the raw socket response from the server, you can access `r.raw`. If you want to do this, make sure you set `stream=True` in your initial request. Once you do, you can do this:

```
>>> r = requests.get('https://api.github.com/events', stream=True)

>>> r.raw
<urllib3.response.HTTPResponse object at 0x101194810>

>>> r.raw.read(10)
'\x1f\x8b\x08\x00\x00\x00\x00\x00\x00\x03'
```

In general, however, you should use a pattern like this to save what is being streamed to a file:

```
with open(filename, 'wb') as fd:
    for chunk in r.iter_content(chunk_size=128):
        fd.write(chunk)
```

Using `Response.iter_content` will handle a lot of what you would otherwise have to handle when using `Response.raw` directly. When streaming a download, the above is the preferred and recommended way to retrieve the content. Note that `chunk_size` can be freely adjusted to a number that may better fit your use cases.

Note: An important note about using `Response.iter_content` versus `Response.raw`. `Response.iter_content` will automatically decode the `gzip` and `deflate` transfer-encodings. `Response.raw` is a raw stream of bytes – it does not transform the response content. If you really need access to the bytes as they were returned, use `Response.raw`.

2.2.7 Custom Headers

If you'd like to add HTTP headers to a request, simply pass in a `dict` to the `headers` parameter.

For example, we didn't specify our user-agent in the previous example:

```
>>> url = 'https://api.github.com/some/endpoint'
>>> headers = {'user-agent': 'my-app/0.0.1'}

>>> r = requests.get(url, headers=headers)
```

Note: Custom headers are given less precedence than more specific sources of information. For instance:

- Authorization headers set with *headers=* will be overridden if credentials are specified in `.netrc`, which in turn will be overridden by the `auth=` parameter.
- Authorization headers will be removed if you get redirected off-host.
- Proxy-Authorization headers will be overridden by proxy credentials provided in the URL.
- Content-Length headers will be overridden when we can determine the length of the content.

Furthermore, Requests does not change its behavior at all based on which custom headers are specified. The headers are simply passed on into the final request.

Note: All header values must be a `string`, bytestring, or unicode. While permitted, it's advised to avoid passing unicode header values.

2.2.8 More complicated POST requests

Typically, you want to send some form-encoded data — much like an HTML form. To do this, simply pass a dictionary to the `data` argument. Your dictionary of data will automatically be form-encoded when the request is made:

```
>>> payload = {'key1': 'value1', 'key2': 'value2'}

>>> r = requests.post("https://httpbin.org/post", data=payload)
>>> print(r.text)
{
  ...
  "form": {
    "key2": "value2",
    "key1": "value1"
  },
  ...
}
```

The `data` argument can also have multiple values for each key. This can be done by making `data` either a list of tuples or a dictionary with lists as values. This is particularly useful when the form has multiple elements that use the same key:

```
>>> payload_tuples = [('key1', 'value1'), ('key1', 'value2')]
>>> r1 = requests.post('https://httpbin.org/post', data=payload_tuples)
>>> payload_dict = {'key1': ['value1', 'value2']}
>>> r2 = requests.post('https://httpbin.org/post', data=payload_dict)
>>> print(r1.text)
{
  ...
  "form": {
```

(continues on next page)

(continued from previous page)

```
    "key1": [
      "value1",
      "value2"
    ]
  },
  ...
}
>>> r1.text == r2.text
True
```

There are times that you may want to send data that is not form-encoded. If you pass in a `string` instead of a `dict`, that data will be posted directly.

For example, the GitHub API v3 accepts JSON-Encoded POST/PATCH data:

```
>>> import json

>>> url = 'https://api.github.com/some/endpoint'
>>> payload = {'some': 'data'}

>>> r = requests.post(url, data=json.dumps(payload))
```

Instead of encoding the `dict` yourself, you can also pass it directly using the `json` parameter (added in version 2.4.2) and it will be encoded automatically:

```
>>> url = 'https://api.github.com/some/endpoint'
>>> payload = {'some': 'data'}

>>> r = requests.post(url, json=payload)
```

Note, the `json` parameter is ignored if either `data` or `files` is passed.

Using the `json` parameter in the request will change the `Content-Type` in the header to `application/json`.

2.2.9 POST a Multipart-Encoded File

Requests makes it simple to upload Multipart-encoded files:

```
>>> url = 'https://httpbin.org/post'
>>> files = {'file': open('report.xls', 'rb')}

>>> r = requests.post(url, files=files)
>>> r.text
{
  ...
  "files": {
    "file": "<censored...binary...data>"
  },
  ...
}
```

You can set the filename, content_type and headers explicitly:

```
>>> url = 'https://httpbin.org/post'
>>> files = {'file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-
↪excel', {'Expires': '0'})}
```

(continues on next page)