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.

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.

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
```

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.

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.

The `br` transfer-encoding is automatically decoded for you if a Brotli library like brotli or brotlicffi is installed.

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))
```

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 `requests.exceptions.JSONDecodeError`. This wrapper exception provides interoperability for multiple exceptions that may be thrown by different python versions and json serialization libraries.

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.

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`.

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. Requests will search for the netrc file at *~/.netrc*, *~/_netrc*, or at the path specified by the *NETRC* environment variable.
- 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.

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": {
    "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))
```

Please note that the above code will NOT add the `Content-Type` header (so in particular it will NOT set it to `application/json`).

If you need that header set and you don't want to encode 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.

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-exc
```

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

If you want, you can send strings to be received as files:

```
>>> url = 'https://httpbin.org/post'
>>> files = {'file': ('report.csv', 'some,data,to,send\nanother,row,to,send\n')}

>>> r = requests.post(url, files=files)
>>> r.text
{
  ...
  "files": {
    "file": "some,data,to,send\\nanother,row,to,send\\n"
  },
  ...
}
```

In the event you are posting a very large file as a `multipart/form-data` request, you may want to stream the request. By default, `requests` does not support this, but there is a separate package which does - `requests-toolbelt`. You should read the toolbelt's documentation for more details about how to use it.

For sending multiple files in one request refer to the advanced section.

> ### Warning
>
> It is strongly recommended that you open files in binary mode. This is because Requests may attempt to provide the `Content-Length` header for you, and if it does this value will be set to the number of *bytes* in the file. Errors may occur if you open the file in *text mode*.

Response Status Codes

We can check the response status code:

```
>>> r = requests.get('https://httpbin.org/get')
>>> r.status_code
200
```

Requests also comes with a built-in status code lookup object for easy reference:

```
>>> r.status_code == requests.codes.ok
True
```

If we made a bad request (a 4XX client error or 5XX server error response), we can raise it with **`Response.raise_for_status()`**:

```
>>> bad_r = requests.get('https://httpbin.org/status/404')
>>> bad_r.status_code
404

>>> bad_r.raise_for_status()
Traceback (most recent call last):
  File "requests/models.py", line 832, in raise_for_status
    raise http_error
requests.exceptions.HTTPError: 404 Client Error
```

But, since our `status_code` for `r` was `200`, when we call `raise_for_status()` we get:

```
>>> r.raise_for_status()
None
```

All is well.

Response Headers

We can view the server's response headers using a Python dictionary:

```
>>> r.headers
{
    'content-encoding': 'gzip',
    'transfer-encoding': 'chunked',
    'connection': 'close',
    'server': 'nginx/1.0.4',
    'x-runtime': '148ms',
    'etag': '"e1ca502697e5c9317743dc078f67693f"',
    'content-type': 'application/json'
}
```

The dictionary is special, though: it's made just for HTTP headers. According to RFC 7230, HTTP Header names are case-insensitive.

So, we can access the headers using any capitalization we want:

```
>>> r.headers['Content-Type']
'application/json'

>>> r.headers.get('content-type')
'application/json'
```

It is also special in that the server could have sent the same header multiple times with different values, but requests combines them so they can be represented in the dictionary within a single mapping, as per RFC 7230:

> A recipient MAY combine multiple header fields with the same field name into one "field-name: field-value" pair, without changing the semantics of the message, by appending each subsequent field value to the combined field value in order, separated by a comma.

Cookies

If a response contains some Cookies, you can quickly access them:

```
>>> url = 'http://example.com/some/cookie/setting/url'
>>> r = requests.get(url)

>>> r.cookies['example_cookie_name']
'example_cookie_value'
```

To send your own cookies to the server, you can use the `cookies` parameter:

```
>>> url = 'https://httpbin.org/cookies'
>>> cookies = dict(cookies_are='working')

>>> r = requests.get(url, cookies=cookies)
>>> r.text
'{"cookies": {"cookies_are": "working"}}'
```

Cookies are returned in a **`RequestsCookieJar`**, which acts like a `dict` but also offers a more complete interface, suitable for use over multiple domains or paths. Cookie jars can also be passed in to requests:

```
>>> jar = requests.cookies.RequestsCookieJar()
>>> jar.set('tasty_cookie', 'yum', domain='httpbin.org', path='/cookies')
>>> jar.set('gross_cookie', 'blech', domain='httpbin.org', path='/elsewhere')
>>> url = 'https://httpbin.org/cookies'
>>> r = requests.get(url, cookies=jar)
>>> r.text
'{"cookies": {"tasty_cookie": "yum"}}'
```

Redirection and History

By default Requests will perform location redirection for all verbs except HEAD.

We can use the `history` property of the Response object to track redirection.

The **`Response.history`** list contains the **`Response`** objects that were created in order to complete the request. The list is sorted from the oldest to the most recent response.

For example, GitHub redirects all HTTP requests to HTTPS:

```
>>> r = requests.get('http://github.com/')

>>> r.url
'https://github.com/'

>>> r.status_code
200

>>> r.history
[<Response [301]>]
```

If you're using GET, OPTIONS, POST, PUT, PATCH or DELETE, you can disable redirection handling with the `allow_redirects` parameter:

```
>>> r = requests.get('http://github.com/', allow_redirects=False)

>>> r.status_code
301

>>> r.history
[]
```

If you're using HEAD, you can enable redirection as well:

```
>>> r = requests.head('http://github.com/', allow_redirects=True)

>>> r.url
'https://github.com/'

>>> r.history
[<Response [301]>]
```

Timeouts

You can tell Requests to stop waiting for a response after a given number of seconds with the `timeout` parameter. Nearly all production code should use this parameter in nearly all requests. Failure to do so can cause your program to hang indefinitely:

```
>>> requests.get('https://github.com/', timeout=0.001)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
requests.exceptions.Timeout: HTTPConnectionPool(host='github.com', port=80): Request
```

> ## Note
>
> `timeout` is not a time limit on the entire response download; rather, an exception is raised if the server has not issued a response for `timeout` seconds (more precisely, if no bytes have been received on the underlying socket for `timeout` seconds). If no timeout is specified explicitly, requests do not time out.

Errors and Exceptions

In the event of a network problem (e.g. DNS failure, refused connection, etc), Requests will raise a **`ConnectionError`** exception.

`Response.raise_for_status()` will raise an **`HTTPError`** if the HTTP request returned an unsuccessful status code.

If a request times out, a **`Timeout`** exception is raised.

If a request exceeds the configured number of maximum redirections, a **`TooManyRedirects`** exception is raised.

All exceptions that Requests explicitly raises inherit from **`requests.exceptions.RequestException`**.

Ready for more? Check out the advanced section.

If you're on the job market, consider taking this programming quiz. A substantial donation will be made to this project, if you find a job through this platform.