[The awesome requests module](http://www.pythonforbeginners.com/python-on-the-web/the-awesome-requests-module/)

12 Oct 2012   | [Web & Internet](http://www.pythonforbeginners.com/category/python-on-the-web/)

Tags: [Modues](http://www.pythonforbeginners.com/tag/modues/) · [Requests](http://www.pythonforbeginners.com/tag/requests/)

What is Requests

The Requests module is a an elegant and simple HTTP library for Python.

What can I do with Requests?

Requests allow you to send HTTP/1.1 requests.

You can add headers, form data, multipart files, and parameters with simple Python dictionaries, and access the response data in the same way.

Note, the notes in this post are taken from Python-Requests.org : <http://docs.python-requests.org/en/latest/>

Requests Installation

To install requests, simply:

$ pip install requests

Or, if you absolutely must:

$ easy\_install requests

Make a Request

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

|  |  |
| --- | --- |
| 1 | >>> r = requests.get('<https://github.com/timeline.json>') |

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

To make a HTTP POST request

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | >>> r = requests.post("<http://httpbin.org/post>")    You can also use other HTTP request types, like PUT, DELETE, HEAD and OPTIONS    >>> r = requests.put("<http://httpbin.org/put>")    >>> r = requests.delete("<http://httpbin.org/delete>")    >>> r = requests.head("<http://httpbin.org/get>")    >>> r = requests.options("<http://httpbin.org/get>") |

Response Content

We can read the content of the server’s response.

Consider the GitHub timeline again:

|  |  |
| --- | --- |
| 1  2  3  4 | >>> import requests  >>> r = requests.get('<https://github.com/timeline.json>')  >>> 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:

|  |  |
| --- | --- |
| 1  2  3 | >>> 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.

Binary Response Content

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

|  |  |
| --- | --- |
| 1  2 | >>> r.content  b'[{"repository":{"open\_issues":0,"url":"https://github.com/... |

JSON Response Content

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

|  |  |
| --- | --- |
| 1  2  3  4 | >>> import requests  >>> r = requests.get('<https://github.com/timeline.json>')  >>> r.json  [{u'repository': {u'open\_issues': 0, u'url': 'https://github.com/... |

In case the JSON decoding fails, r.json simply returns None.

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 content-type in the previous example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | >>> import json  >>> url = '<https://api.github.com/some/endpoint>'  >>> payload = {'some': 'data'}  >>> headers = {'content-type': 'application/json'}    >>> r = requests.post(url, data=json.dumps(payload), headers=headers) |

Response Status Codes

We can check the response status code:

|  |  |
| --- | --- |
| 1  2  3 | >>> r = requests.get('<http://httpbin.org/get>')  >>> r.status\_code  200 |
| 1  2  3 | # Requests also comes with a built-in status code lookup object for easy reference:  >>> r.status\_code == requests.codes.ok  True |

|  |  |
| --- | --- |
| 1  2  3  4  5 | # If we made a bad request (non-200 response),  # we can raise it with Response.raise\_for\_status():  >>> bad\_r = requests.get('<http://httpbin.org/status/404>')  >>> bad\_r.status\_code  404 |

Response Headers

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | >>> r.headers  {      'status': '200 OK',      'content-encoding': 'gzip',      'transfer-encoding': 'chunked',      'connection': 'close',      'server': 'nginx/1.0.4',      'x-runtime': '148ms',      'etag': '"e1ca502697e5c9317743dc078f67693f"',      'content-type': 'application/json; charset=utf-8'  } |

HTTP Headers are case-insensitive, so we can access the headers using any

capitalization we want:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | >>> r.headers['Content-Type']  'application/json; charset=utf-8'    >>> r.headers.get('content-type')  'application/json; charset=utf-8'    # If a header doesn’t exist in the Response, its value defaults to None:  >>> r.headers['X-Random']  None |

Cookies

If a response contains some Cookies, you can get quick access to them:

|  |  |
| --- | --- |
| 1  2  3  4  5 | >>> url = '<http://httpbin.org/cookies/set/requests-is/awesome>'  >>> r = requests.get(url)    >>> r.cookies['requests-is']  'awesome' |
| 1  2  3  4  5  6  7  8  9 | # To send your own cookies to the server, you can use the cookies parameter:    >>> url = '<http://httpbin.org/cookies>'  >>> cookies = dict(cookies\_are='working')    >>> r = requests.get(url, cookies=cookies)    >>> r.text  '{"cookies": {"cookies\_are": "working"}}' |

Basic Authentication

Many web services require authentication.

There are many different types of authentication, but the most common is HTTP

Basic Auth.

Making requests with Basic Auth is extremely simple:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | from requests.auth import HTTPBasicAuth  requests.get('<https://api.github.com/user>', auth=HTTPBasicAuth('user', 'pass'))  <response [200]="">    # Due to the prevalence of HTTP Basic Auth,  # requests provides a shorthand for this authentication method:    requests.get('<https://api.github.com/user>', auth=('user', 'pass'))  <response [200]="">  </response></response> |

Providing the credentials as a tuple in this fashion is functionally equivalent to the  
HTTPBasicAuth example above.

Digest Authentication

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | # Another popular form of web service protection is Digest Authentication:    >>> from requests.auth import HTTPDigestAuth    >>> url = '<http://httpbin.org/digest-auth/auth/user/pass>'    >>> requests.get(url, auth=HTTPDigestAuth('user', 'pass'))  <response [200]="">  </response> |

Redirection and History

Requests will automatically perform location redirection while using the GET

and OPTIONS verbs.

GitHub redirects all HTTP requests to HTTPS.

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

Let’s see what Github does:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | >>> import requests  >>> r = requests.get("[http://github.com](http://github.com/)")  >>> r.url  u'<https://github.com/>'  >>> r.status\_code  200  >>> r.history  [<response [301]="">]  >>>  </response> |

The Response.history list contains a list of the Request objects that were

created in order to complete the request.

The list is sorted from the oldest to the most recent request.

If you’re using GET or OPTIONS, you can disable redirection handling with the

allow\_redirects parameter:

|  |  |
| --- | --- |
| 1  2  3  4  5 | >>> r = requests.get('[http://github.com](http://github.com/)', allow\_redirects=False)  >>> r.status\_code  301  >>> r.history  [] |

Timeouts

You can tell requests to stop waiting for a response after a given number of

seconds with the timeout parameter:

|  |  |
| --- | --- |
| 1  2  3  4  5 | >>> requests.get('[http://github.com](http://github.com/)', timeout=0.001)  Traceback (most recent call last):    File "<stdin>", line 1, in <module>  requests.exceptions.Timeout: Request timed out.  </module></stdin> |

Errors and Exceptions

In the event of a network problem (e.g. DNS failure, refused connection, etc),

Requests will raise a ConnectionError exception.

In the event of the rare invalid HTTP response,

Requests will raise an HTTPError exception.

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.

You can refer to Configuration API Docs for immediate raising of

HTTPError exceptions via the danger\_mode option or have Requests

catch the majority of requests.exceptions.

RequestException exceptions with the safe\_mode option.