Feb. 11, 2013

Requests Web & Internet

Using the Requests Library in Python

First things first, let’s introduce you to Requests.

What is the Requests Resource?

Requests is an Apache2 Licensed HTTP library, written in Python. It is designed to be used by humans to interact with the language. This means you don’t have to manually add query strings to URLs, or form-encode your POST data. Don’t worry if that made no sense to you. It will in due time.

What can Requests do?

Requests will allow you to send HTTP/1.1 requests using Python. With it, you can add content like headers, form data, multipart files, and parameters via simple Python libraries. It also allows you to access the response data of Python in the same way.

In programming, a library is a collection or pre-configured selection of routines, functions, and operations that a program can use. These elements are often referred to as modules, and stored in object format.

Libraries are important, because you load a module and take advantage of everything it offers without explicitly linking to every program that relies on them. They are truly standalone, so you can build your own programs with them and yet they remain separate from other programs.

Think of modules as a sort of code template.

To reiterate, Requests is a Python library.

How to Install Requests

The good news is that there are a few ways to install the Requests library. To see the full list of options at your disposal, you can view the official install documentation for Requests here.

You can make use of pip, easy_install, or tarball.

If you’d rather work with source code, you can get that on GitHub, as well.

For the purpose of this guide, we are going to use pip to install the library.

In your Python interpreter, type the following:

```
pip install requests
```

Importing the Requests Module

To work with the Requests library in Python, you must import the appropriate module. You can do this simply by adding the following code at the beginning of your script:

```
import requests
```

Of course, to do any of this – installing the library included – you need to download the necessary package first and have it accessible to the interpreter.

Making a Request

When you ping a website or portal for information this is called making a request. That is exactly what the Requests library has been designed to do.

To get a webpage you would do something like the following:

```
r = request.get(‘https://github.com/timeline.json’)
```

Working with Response Code

Before you can do anything with a website or URL in Python, it’s a good idea to check the current status code of said portal. You can do this with the dictionary look-up object.

```
r = requests.get('https://github.com/timeline.json')
r.status_code
>>200

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

requests.codes['temporary_redirect']
>>> 307

requests.codes.teapot
>>> 418

requests.codes['o/']
>>> 200
```

Get the Content

After a web server returns a response, you can collect the content you need. This is also done using the get requests function.

```
import requests
r = requests.get('https://github.com/timeline.json')
print r.text

# The Requests library also comes with a built-in JSON decoder,
# just in case you have to deal with JSON data

import requests
r = requests.get('https://github.com/timeline.json')
print r.json
```

Working with Headers

By utilizing a Python dictionary, you can access and view a server's response headers. Thanks to how Requests works, you can access the headers using any capitalization you'd like.

If you perform this function but a header doesn't exist in the response, the value will default to None.

```
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'
}

r.headers['Content-Type']
>>>'application/json; charset=utf-8'

r.headers.get('content-type')
>>>'application/json; charset=utf-8'

r.headers['X-Random']
>>>None

# Get the headers of a given URL
resp = requests.head("http://www.google.com")
print resp.status_code, resp.text, resp.headers
```

Encoding

Requests will automatically decade any content pulled from a server. But most Unicode character sets are seamlessly decoded anyway.

When you make a request to a server, the Requests library make an educated guess about the encoding for the response, and it does this based on the HTTP headers. The encoding that is guessed will be used when you access the r.text file.

Through this file, you can discern what encoding the Requests library is using, and change it if need be. This is possible thanks to the *r.encoding* property you'll find in the file.

If and when you change the encoding value, Requests will use the new type so long as you call r.text in your code.

```
print r.encoding
>> utf-8

>>> r.encoding = ‘ISO-8859-1’
```

Custom Headers

If you want to add custom HTTP headers to a request, you must pass them through a dictionary to the headers parameter.

```
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)
```

Redirection and History

Requests will automatically perform a location redirection when you use the GET and OPTIONS verbs in Python.

GitHub will redirect all HTTP requests to HTTPS automatically. This keeps things secure and encrypted.

You can use the history method of the response object to track redirection status.

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

r.status_code
>>> 200

r.history
>>> []
```

Make an HTTP Post Request

You can also handle post requests using the Requests library.

```
r = requests.post(http://httpbin.org/post)
```

But you can also rely on other HTTP requests too, 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")
```

You can use these methods to accomplish a great many things. For instance, using a Python script to create a GitHub repo.

```
import requests, json

github_url = "https://api.github.com/user/repos"
data = json.dumps({'name':'test', 'description':'some test repo'})
r = requests.post(github_url, data, auth=('user', '*****'))

print r.json
```

Errors and Exceptions

There are a number of exceptions and error codes you need to be familiar with when using the Requests library in Python.

- If there is a network problem like a DNS failure, or refused connection the Requests library will raise a ConnectionError exception.
- With invalid HTTP responses, Requests will also raise an HTTPError exception, but these are rare.
- If a request times out, a Timeout exception will be raised.
- If and when a request exceeds the preconfigured number of maximum redirections, then a TooManyRedirects exception will be raised.

Any exceptions that Requests raises will be inherited from the requests.exceptions.RequestException object.

You can read more about the Requests library at the links below.

http://docs.python-requests.org/en/latest/api/

http://pypi.python.org/pypi/requests

http://docs.python-requests.org/en/latest/user/quickstart/

http://isbullsh.it/2012/06/Rest-api-in-python/#requests

Recommended Python Training – DataCamp

For Python training, our top recommendation is DataCamp.

Datacamp provides online interactive courses that combine interactive coding challenges with videos from top instructors in the field.

Datacamp has beginner to advanced Python training that programmers of all levels benefit from.