Amit Verma (axv68)
Shujah Ahmad (saa3947)

Tutorial 9: Requests Python Library

About *Requests* Library:

Requests is a modern, human friendly HTTP python library that is licensed under Apache 2.0. When an HTTP request is made, a response object is returned containing data associated with the respective request. Some of the common requests used from this library are GET and POST, but many others like HEAD, PATCH, DELETE, and OPTIONS are also supported. This library offers several beneficial features such as Connection Timeouts and Automatic Decompression of Content. According to Github, *Requests* is one of the most downloaded Python libraries, having over 14M downloads.

Competitors/Alternatives:

The main competitors and alternatives to the Python Requests library are urllib, urllib2, urllib3, and httplib2. In general, the Requests library is known for its security , its multitude of applications, and ease of use. The Requests library usually requires less code to perform the same actions as its other competitors, and is known in the industry as being easier to use and learn than its competitors. One of the biggest advantages of Requests over its competitors is that it allows the user to pass in Request objects automatically as parameters. However, there are specific cases where the alternatives outperform it. For instance, if a user wants to do have support for advanced caching, then httplib2 would be better to use. In addition, the urllib libraries are better and faster than the Requests library for supporting requests with a lot of data.

Use Cases:

Below is a list of potential use cases that would require the *Requests* Python library:

- Fetch and receive data from a website
- To test a website
- Make authenticated requests
- Use to easily inspect data
- Send/Receive Payments
- Making an online schedule
- Send/Receive emails
- File Transfer
- Using Map Technologies (i.e. GPS)
- Error Handling
- Security
- Desire to write less code

Below are times when one shouldn't use the *Requests* Library:

Amit Verma (axv68)
Shujah Ahmad (saa3947)

Tutorial 9: Requests Python Library

- Big Requests
- Specifying/Adding/Setting Headers
- When using Google App Engine
- Advanced Caching Support

Installation and Setup:

Before installing: Make sure you have python3 already installed on your machine. If not, you can download the latest version of python3 by going to: https://www.python.org/downloads/. After downloading, restart your computer and run `pip3 -h` in your terminal to ensure that python3 has been successfully installed.

Installation: Run the following command below in your terminal/shell in order to download the *Requests* library

```
pip3 -m install requests
```

Once you have installed the library, you are able to use it in your python application by putting `import requests` at the top of your python file.

Github Example:

Create a python file in an IDE of your choice and call it 'requests_lirbary_demo.py'. This will be the file in which you will put your python code for this tutorial.

At the top of your python file, import the requests library by putting `import requests` at the top of your python file.

1. GET Requests

GET requests are used in order to retrieve data from the server. To make a GET request, we will use `requests.get()` , in which the parameter passed will be a URL. GET requests are typically written in the following format:

```
requests.get(url, params={key: value}, args)
```

Write the the following python code in your file:

```
response = requests.get('https://api.github.com')
```

Amit Verma (axv68)
Shujah Ahmad (saa3947)

Tutorial 9: Requests Python Library

```
print(response)
```

The code above exemplifies a simple GET request, and by running the file, a status code should be printed on your console saying `<Response [200]>`. The '.get' function will take a URL and return an object back into the variable 'response'. When printing the returned object 'response', by default you are shown the status code. A *status code* informs you about the status of the request you made, giving you a general understanding of whether your request was successful or not.

In our case, from the example, we were returned a status code of 200, which means that our request was successful. A common unsuccessful status code is 404, which means that the page is not found. A simple notion to follow is that if your status code is in the 200s, then your requests will be successful. On the other hand, if your requests are in the 400s, then your requests will not be successful. The Requests library helps you evaluate status codes easily with conditional expressions as it denotes status codes in the 200 to be 'True', and codes in the 400s to be 'False'. By pasting the following code below in your python file and running it, you will see that `Success!` is printed on your console.

```
if response:
    print('Success!')
else:
    print('An error has occurred.')
```

2. Reponses and Contents: Methods used to read GET Request response

After getting a response object (also known as a payload) from your GET request, there are several different ways (or methods) in which you can parse the object to get certain data. This payload data is pretty structured in a JSON format, making it very easy to read. Keep in mind that the following methods below are also usable to view data from POST requests too.

Method 1: If you want to view the data in bytes, you can simply write `print(response.content)` in your file.

Method 2: To easily convert the payload into a string, you can use the .text method, which will use a character encoding like UTF-8. To do this, you can write `print(response.text)`

Method 3: To easily get dictionary format of the JSON structured data, you can write `print(response.json())`

3. POST Requests

Amit Verma (axv68)
Shujah Ahmad (saa3947)

Tutorial 9: Requests Python Library

POST requests are used to submit data to be processed to the server. A POST request will use the .post() method and pass in a data parameter. POST requests are typically written in the following format:

```
requests.post(url, data={key: value}, json={key: value}, args)
```

The following code below will show an example of a simple POST request:

```
payload = {'username': 'EE461L', 'password': 'nemo'}
postrequest = requests.post('https://httpbin.org/post', data=payload)
print(postrequest.text)
```

In this format, our 'payload' is the data we want to be sent to the server. When making a POST request, we always pass a parameter called 'data' and set it equal to the payload we make. By using the .text method, we are able to see that our data (containing a username and password field) did in fact did get posted as you will see following JSON response on your console:

```
 "form": {
    "mascot": "nemo",
    "name": "EE461L"
  }
```

In this specific example, we happen to be passing form data, but we can pass in different types of data too as it is. If you are trying to ever pass in certain form data to a specific webpage, you will need to look at the HTML source code of the URL to understand what values that form data will expect.

Now, if we could even make a dictionary with our POST request data and then access the form field to ensure that our data is posted:

```
pr_dict = postrequest.json()
print(pr_dict['form'])
```

Upon running this code in your file, you should see the following output in your console: `{u'name': u'EE461L', u'mascot': u'nemo'}`

4. Authentication

Authentication related processes use GET requests that are passed an authorization header dubbed 'auth'. The general format looks like the following:

```
request = requests.get('<URL>', auth=(<username>, <password>)
```

Amit Verma (axv68)
Shujah Ahmad (saa3947)

Tutorial 9: Requests Python Library

For the sake of this next example, put `from getpass import getpass` at the top of your file. Then, paste the following example in your python file and **edit the username section with your Github ID where it says** `<YOUR GITHUB USERNAME>`:

```
request = requests.get('https://api.github.com/user', auth=('<YOUR GITHUB USERNAME>', getpass())
```

In this example, we'll see that we are passing a username (which is your actual Github ID) and then a method 'getpass()' as the password. When running this code, this method will prompt you to enter a password on the console. Upon entering your password and clicking enter, the password will be checked and return a status code to inform you on whether or not your login was successful. If your login was successful, meaning that your username and password are valid, then you will receive a status code of 200. Otherwise, you will receive a status code that is over 400.

NOTE
Typically, the 'getpass()' method would be replaced with an actual password, which would result in you not having to enter a password on the console.

5. Timeouts

Timeouts are a good way of understanding how long your system may need to wait in order to get a response from the server. If your requests are very heavy and consume a lot of time, then requests to your website or service could back up and result in a poor user experience.

By default, when making a request, there is not a specified threshold wait time to receive a response. So, in order to set a timeout, you will need to add that as a parameter to within the request you are making. The 'timeout' parameter takes a number, which is wait time (in seconds) for the response to the request.

For example, run the following code in your file:

```
r = requests.get('https://api.github.com', timeout=1)
print(r)
```

The status code returned here should be 200 as the response to load the website shouldn't take more than a second. Now, if we changed the timeout to a number like .004 and rerun the same code, then we will get an error that says `'Connection to api.github.com timed out. (connect timeout=0.0004)'`

Amit Verma (axv68)
Shujah Ahmad (saa3947)

Tutorial 9: Requests Python Library

Oftentimes, you can even catch timeouts as exceptions when testing your code. In your file, `from requests.exceptions import Timeout` at the top of your file. In doing so, below is a small example of how you can handle a timeout exception.

```
try:
    response = requests.get('https://api.github.com', timeout=1)
except Timeout:
    print('The request timed out')
else:
    print('The request did not time out')
```

In this second example, you can see how the exception is handled by printed output when the timeout time varies. When testing your website, you can use such a test to see how heavy your requests may be taking.

Wrapping Up

Now, the five features covered in this tutorial are few of many that can be used within the Requests library. If you would like, click on this link to see the code from this tutorial. If you are interested in learning more about this library, please visit the following links below:

- https://realpython.com/python-requests/#authentication
- https://requests.readthedocs.io/en/master/