The Python Requests Library

Up to now, we've seen how we can serialize the data that we have in our programs and turn it into a format that we can store on disk. Once the data is stored, another process can open up those files, de-serialize them, and go from there.

This works, but only if the other process has access to the same filesystem you used to store your data. What if you wanted to send a message to another computer on another network? HTTP to the rescue!

Remember that ***HTTP (HyperText Transfer Protocol)*** is the protocol of the world-wide web. When you visit a webpage with your web browser, the browser is making a series of ***HTTP requests*** to web servers somewhere out on the Internet. Those servers will answer with ***HTTP responses***. This is also how we're going to send and receive messages with web applications from our code.

The Python Requests library makes it super easy to write programs that send and receive HTTP. Instead of having to understand the HTTP protocol in great detail, you can just make very simple HTTP connections using Python objects, and then send and receive messages using the methods of those objects. Let's look at an example:

```
>>> import requests
>>> response = requests.get('https://www.google.com')
```

That's it! That was a basic request for a web page! We used the Requests library to make a ***HTTP GET*** request for a specific ***URL, or Uniform Resource Locator***. The URL tells the Requests library the name of the resource (**www.google.com**) and what protocol to use to get the resource (**https://**). The result we get is an object of type requests.Response.

Alright, now what did the web server respond with? Let's take a look at the first 300 characters of the Response.text:

```
>>> print(response.text[:300])
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="de"><head><meta content="text/html; charset=UTF-8" http-
```

Now, it might be hard for you to read the HTML (HyperText Markup Language) that was returned in this response, but your web browser knows just how to turn that into a familiar-looking web page.

Even with this simple example, the Requests module has done a whole lot of work for us! We didn't have to write any code to find the web server, make a network connection, construct an HTTP message, wait for a response, or decode the response. Not that HTML can't be messy enough on its own, but let's look at the first bytes of the ***raw*** message that we received from the server:

```
>>> response = requests.get('https://www.google.com', stream=True)
>>> print(response.raw.read()[:100])
b'\x1f\x8b\x08\x00\x00\x00\x00\x00\x02\xff\xc5Z\xdbz\x9b\xc8\x96\xbe\xcfS`\xf2\xb5-\xc6X\x02$t\xc28\xe3v\xdc\xdd\xee\xce\xa9\xb7\xdd;
```

What's all that? The response was ***compressed*** with ***gzip***, so it had to be ***decompressed*** before we could even read the text of the HTML. One more thing that the Requests library handled for us!

The requests.Response object also contains the exact request that was created for us. We can check out the headers stored in our object to see that the Requests module told the web server that it was okay to compress the content:

```
>>> response.request.headers['Accept-Encoding']
'gzip, deflate'
```

And then the server told us that the content had actually been compressed.

```
>>> response.headers['Content-Encoding']
'gzip'
```

And all this happened by default, without us having to do anything special to make it work. Amazing, right?

Completed **Go to next item**

Like **Dislike** **Report an issue**