Theory: Requests: retrieving data

21 minutes 0 / 5 problems solved

This topic is an introduction to `requests`, an elegant and simple HTTP library for Python.

Hypertext Transfer Protocol (HTTP) is arguably the most popular application protocol used on the Internet. It allows for communication between HTTP clients (e.g., web browsers) and HTTP servers (web servers) by sending text messages: the client sends a **request** message to the server which, in turn, returns a **response** message.

Python `requests` allows you to send all kinds of requests and get the responses to them in an easy and intuitive way. Don't forget that it's not in the standard library, and therefore you might need to install it by typing `pip install requests` in your command line.

To start using `requests` in your code, import the library:

```
import requests
```

§1. Making a GET request

The **GET** request is used to retrieve information from the given server using a URL. For example, whenever you enter a URL in the address box of your browser, it translates it into a **GET** request message and sends it to the server.

Imagine we need to get the main page of the official website of the `requests` library. We can do so with the help of `requests.get(url)` as follows:

```
r = requests.get('https://requests.readthedocs.io/en/master/')
print(r)

# <Response [200]>
```

This returns a **response** object `r` containing all the necessary information about the response of the server. Note that 200 is a standard HTTP code indicating that the request succeeded, while code 404 means that the resource you were looking for was not found. You can explicitly access the response code in the `status_code` attribute of the response object:

```
print(r.status_code)

# 200
```

If you use a `response` object in an `if` statement, it will evaluate to `True` if the status code starts with the digits 2 (request was accepted) or 3 (redirection), and `False` otherwise:

```
if r:
    print('Success!')
else:
    print('Fail')

# Success!
```

To read the content of the server's response, look at the `text` property:

```
r.text

# '\n<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"\n ...
# ...
```

Current topic:

Requests: retrieving data

Topic depends on:

Stage 1 HTTP messages

Stage 1 Dictionary 6

Stage 1 Declaring a function 10

Stage 1 Pip

Topic is required for:

BeautifulSoup

Table of contents:

↑ Requests: retrieving data

§1. Making a GET request

§2. Query parameters

§3. Conclusions

Feedback & Comments

Note that `requests` automatically decodes the content of the response from the server. You can find out what encoding is being used and change it, if necessary, using the `encoding` property:

```
r.encoding

# 'ISO-8859-1'
```

Other useful information, e.g., content type, is stored in the response headers. To view them, access `.headers`:

```
r = requests.get('https://requests.readthedocs.io/en/master/')
print(r.headers)

# {'Transfer-Encoding': 'chunked', 'Content-Type': 'text/html', 'Content-Encoding': 'gzip',
#  'Last-Modified': 'Thu, 09 Jan 2020 17:56:10 GMT', 'ETag': 'W/"5e17693a-b523"', 'Vary':
#  'Accept-Encoding', 'Server': 'nginx', 'X-Subdomain-TryFiles': 'True', 'X-Served': 'Nginx',
#  'X-Deity': 'web03', 'Strict-Transport-Security': 'max-age=31536000; includeSubDomains',
#  'Date': 'Thu, 16 Jan 2020 13:31:20 GMT'}
```

A dictionary-like object is returned, so you can access the header value you need by its key. Note that the headers are case-insensitive, meaning you don't have to worry about their capitalization:

```
r.headers['Content-Type']

# 'text/html'

# This work just fine as well
r.headers['content-type']

# 'text/html'
```

§2. Query parameters

A **query string** is a convention for appending key-value pairs to a URL. It's separated from the standard URL with a question mark sign `?` and contains key-value pairs. Each key is separated from the value by an equality sign `=`, while the pairs are separated by an ampersand `&`.

Query strings can include fields added to a base URL by the browser or other client applications. How these parameters are used is up to the server-side application. For example, https://www.python.org/search/ is a search page of the official Python website. If you search for 'requests' there, the results will be displayed on the page with the URL https://www.python.org/search/?q=requests.

When using `requests`, there's no need to manually add query strings to your URLs. The library allows you to provide these arguments as a dictionary of strings using the `params` keyword argument when making a request:

```
# The dictionary of the query parameters
params = {'q': 'requests'}


# This request will get the page with the results of the search for 'requests'
# on the official Python website:
r = requests.get('http://python.org/search', params=params)
```

If you need to send similar requests multiple times in your project, it makes sense to define a special function for that. For example, `google_search(query, num)` returns a URL to the page containing `num` Google search results for a given `query` :

```
def google_search(query, num):
    r = requests.get('https://google.com/search',
                     params={'q': query, 'num': num})
    return r.url


print(google_search('python', 1))

# https://www.google.com/search?q=python&num=1
```

Unfortunately, a lot of requests to Google can lead to banning. Try to be attentive when sending multiple requests!

§3. Conclusions

- `requests` is a Python library for making HTTP requests.
- GET request `requests.get()` is used to retrieve the data from the server.
- To provide additional parameters to the server with your GET request, use a query string.
- A query string can be passed to `requests.get()` as a dictionary of key-value pairs.

Report a typo

131 users liked this theory. **2** didn't like it. **What about you?**

Start practicing

Comments (4) Hints (0) Useful links (0) Show discussion