FREE eBook

LEARNING python-requests

Free unaffiliated eBook created from
Stack Overflow contributors.

#python-requests

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: python-requests

It is an unofficial and free python-requests ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official python-requests.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with python-requests

Remarks

HTTP for Humans

Requests is the only Non-GMO HTTP library for Python, safe for human consumption.

Requests allows you to send organic, grass-fed `HTTP/1.1` requests, without the need for manual labor. There's no need to manually add query strings to your URLs, or to form-encode your POST data. Keep-alive and HTTP connection pooling are 100% automatic, powered by `urllib3`, which is embedded within Requests.

The power of Requests:

```
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
u'{"type":"User"...'
>>> r.json()
{u'private_gists': 419, u'total_private_repos': 77, ...}
```

Examples

Installation or Setup

`python-requests` is available on PyPI, the Python Package Index, which means it can be installed through pip:

```
pip install requests
```

Up-to-date source code can be found on the requests GitHub repository

If you wish to install it from source, you can do this by either cloning the GitHub repository:

```
git clone git://github.com/kennethreitz/requests.git
```

Or by getting the tarball (`-o` writes the output to file; `-L` follows redirects):

```
curl -OL https://github.com/kennethreitz/requests/tarball/master
```

Then you can install it by executing the `setup.py`

```
python setup.py install
```

However you installed it, you can start using it by importing the usual way

```
>>> import requests
>>> requests.get('http://stackoverflow.com')
```

GET requests

`requests.get()` creates a GET request:

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

Pass in query parameters as a dictionary to the `params` argument:

```
response = requests.get('https://example.com/', params={"a": 1, "b": 2})
```

For GET requests that might require basic authentication, you can include the `auth` paramter as follows:

```
response = requests.get('https://api.github.com/user', auth=('user', 'pass'))
```

POST requests

POST requests are made with the `request.post()` method.

If you need to send a web form request as a POST body, pass in a dictionary with key-value pairs as the `data` argument; `requests` will encode these to a `application/x-www-form-urlencoded` mimetype body:

```
r = requests.post('https://github.com/', data={"a": 1, "b": 2})
```

If you need to POST a json payload, you can use `json=`. This will automatically set the Content-Type header to `application/json`

```
r = requests.post('https://github.com/', data={"a": 1, "b": 2})
```

Other request methods

The `requests` module has top-level functions for most HTTP methods:

```
r = requests.put('https://example.com/', data=put_body)
```

```
r = requests.delete('https://example.com/')
r = requests.head('https://example.com/')
r = requests.options('https://example.com/')
r = requests.patch('https://example.com/', data=patch_update)
```

Reading the response

```
response = requests.get("https://api.github.com/events")
text_resp = response.text
```

JSON response: for json-formatted responses the package provides a built-in decoder

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

This method will raise a `ValueError` in case of empty response or unparseable content.

Reading status codes

The attribute `status_code` contains the status code of the response

```
good_req = requests.get('https://api.github.com/events')
code_200 = good_req.status_code

notfound_req = requests.get('https://api.github.com/not_found')
code_404 = notfound_req.status_code
```

`requests.codes.__dict__` will provide a list of available http status codes.

It is possible to user `raise_for_status` to check if the status_code was 4xx or 5xx and raise a corresponding exception in that case.

```
good_req = requests.get('https://api.github.com/events')
good_req.raise_for_status()
# is a 200 status code so nothing happens

notfound_req = requests.get('https://api.github.com/not_found')
notfound_req.raise_for_status()
# raises requests.exceptions.HTTPError: 404 Client Error
```

Read Getting started with python-requests online: https://riptutorial.com/python-requests/topic/1115/getting-started-with-python-requests

Chapter 2: Automating login using Requests over Single Sign On

Examples

Example of accessing authenticated pages using requests

Sometimes we have requirement of parsing pages, but doing so requires you to be an authorised user. Here is an example which shows you how to do in oracle sign in.

```
import sys
import requests
import json
from bs4 import BeautifulSoup


def mprint(x):
    sys.stdout.write(x)
    print
    return


headers = {'User-Agent': 'Mozilla/5.0 (X11; Linux i686; rv:7.0.1) Gecko/20100101
Firefox/7.0.1'}

mprint('[-] Initialization...')
s = requests.session()
s.headers.update(headers)
print 'done'


mprint('[-] Gathering JSESSIONID..')

# This  should redirect us to the login page
# On looking at the page source we can find that
# in the submit form 6 values are submitted (at least at the time of this script)
# try to take those values out using beautiful soup
# and then do a post request. On doing post https://login.oracle.com/mysso/signon.jsp
# we will be given message we have the data which is more than necessary
# then it will take us to the form where we have to submit data here
# https://login.oracle.com/oam/server/sso/auth_cred_submit
# once done we are signed in and doing and requests.get(url) will get you the page you want.

r = s.get("company's local url- a link which requires authentication")
if r.status_code != requests.codes.ok:
  print 'error'
  exit(1)
print 'done'

c = r.content
soup = BeautifulSoup(c,'lxml')
svars = {}

for var in soup.findAll('input',type="hidden"):
```

```
    svars[var['name']] = var['value']


s = requests.session()
r = s.post('https://login.oracle.com/mysso/signon.jsp', data=svars)

mprint('[-] Trying to submit credentials...')
inputRaw = open('credentials.json','r')
login = json.load(inputRaw)


data =  {
        'v': svars['v'],
        'OAM_REQ': svars['OAM_REQ'],
        'site2pstoretoken': svars['site2pstoretoken'],
        'locale': svars['locale'],
        'ssousername': login['ssousername'],
        'password': login['password'],
}

r = s.post('https://login.oracle.com/oam/server/sso/auth_cred_submit', data=data)

r = s.get("company's local url- a link which requires authentication")
# dumping the html page to html file
with open('test.html','w') as f:
    f.write(r.content)
```

credentials.json as mentioned in the code is as follows:

```
{
  "ssousername":"example@oracle.com",
  "password":"put your password here"
}
```

Link to github - gist

Read Automating login using Requests over Single Sign On online: https://riptutorial.com/python-requests/topic/6240/automating-login-using-requests-over-single-sign-on

Chapter 3: Django Framework

Examples

Installation & Setup

Django is a full stack framework for web development. It powers some of the most popular websites on the Internet.

To install the framework; use the `pip` tool:

```
pip install django
```

If you are installing this on OSX or Linux, the above command may result in a permission error; to avoid this error, install the package for your user account or use a virtual environment:

```
pip install --user django
```

Once it is installed - you will have access to `django-admin` bootstrapping tool, which will create a directory with some defaults to start development.

```
django-admin startproject myproject
```

This will create a directory *`myproject`* with the default project layout.

Django Core Concepts

Django is a full stack, feature rich web development framework. It bundles a lot of functionality together to provide a common, quick and productive experience for web developers.

Django projects consist of common settings, and one or more *applications*. Each application is a set of functionality along with dependencies (such as templates and models) that are bundled together as Python modules.

The django bootstrapping script automatically creates a settings file for your project, with most common features enabled.

This concept of applications allows easy plug-and-play of functionality, and there is a large library of applications available to handle most common tasks. This concept of applications is fundamental to django; a lot of the built-in functionality (such as user authentication and the admin site) are simply django apps.

To create your first application, from within the project directory:

```
python manage.py startapp yourapp
```

yourapp is the name of your custom application.

Each application allows you to develop:

1. A series of *views* - these are pieces of code that are executed in response to a request.
2. One or more *models*; which are an abstraction to databases. These allow you to define your objects as Python objects, and the built-in ORM provides a friendly API to storing, retrieving and filtering objects from databases.
3. Closely related to models are *migrations* which are scripts that are generated to provide a consistent and reliable method of applying changes in your models, to the database.
4. A set of *urls* that the application will respond to.
5. One or more admin classes; to customize how the application behaves in the built-in django admin application.
6. Any tests that you may write.

Core Concepts - Views

A `view` is any piece of code that responds to a request and returns a response. Views normally return templates along with a dictionary (called the *context*) which usually contains data for placeholders in the template. In django projects, views are located in the `views.py` module of applications.

The simplest view, returns a direct response:

```
from django.http import HttpResponse

def simple_view(request):
   return HttpResponse('<strong>Hello World</strong>')
```

However, most views utilize a template:

```
from django.shortcuts import render

def simple_template_view(request):
    return render(request, 'some_template.html')
```

A template is simply any file, and it can optionally contain special markup for added functionality; what this means is that django views can return any kind of response, not just HTML.

Core Concepts - Templates

In django, a template is simply a file that contains special tags which may be replaced by data from the view.

The canonical template example would be:

```
<strong>Hello {{ name }}, I am a template!</strong>
```

Here, the string `{{ name }}` identifies a placeholder that may be replaced by a context.

To render this template from a view, we can pass in the value as a dictionary:

```
from django.shortcuts import render

def simple_view(request):
    return render(request, 'template.html', {'name': 'Jim'})
```

Once this view is rendered, the resulting HTML will be **Hello Jim, I am a template!**.

Core Concepts - URLs

In django, there is a url mapper which maps URLs to specific functions (views) which return responses. This strict separation between the file system layout and the URL layout allows great flexibility when writing applications.

All url patterns are stored in one or more `urls.py` files, and there is a master `urls.py` file which is read by django first.

Django parses the patterns in the order they are written, and stops when it finds a match to the URL being requested by the user. If no matches are found, an error is raised.

In debug mode (activated by setting `DEBUG = True` in `settings.py`), django will print out a detailed error message when a url requested doesn't match any patterns. In production, however, django will display a normal 404 message.

A url pattern consists of a Python regular expression, followed by a *callable* (a method or function) to be called when that pattern is matched. This function must return a HTTP response:

```
url(r'/hello$', simple_view)
```

Read Django Framework online: https://riptutorial.com/python-requests/topic/6579/django-framework

Chapter 4: Files

Parameters

Parameters	Function
file	JSON List of paths to the files.
content_type	MIME Types
headers	HTTP Headers

Remarks

The `r` variable in the examples contains the full binary data of whatever file you're sending.

Examples

Simple File Upload

```
url = 'http://your_url'
files = {'file': open('myfile.test', 'rb')}
r = requests.post(url, files=files)
```

File Upload w/ Manual Params

```
url = 'http://httpbin.org/post'
files = {'file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-excel',
{'Expires': '0'})}
r = requests.post(url, files=files)
```

Sending Strings as FIles

```
url = 'http://httpbin.org/post'
files = {'file': ('report.csv', 'some,data,to,send\nanother,row,to,send\n')}
r = requests.post(url, files=files)
r.text
```

Read Files online: https://riptutorial.com/python-requests/topic/5929/files

Chapter 5: Sending and receiving JSON

Examples

POSTing JSON

To POST a JSON body, pass in a Python data structure to the `json` argument; here a dictionary is posted but anything that can be encoded to JSON will do:

```
import requests

# Create a dictionary to be sent.
json_data = {'foo': ['bar', 'baz'], 'spam': True, 'eggs': 5.5}

# Send the data.
response = requests.post(url='http://example.com/api/foobar', json=json_data)
print("Server responded with %s" % response.status_code)
```

`requests` takes care of encoding to JSON for you, and sets the `Content-Type` to `application/json`.

Receiving JSON in a response

When a response contains valid JSON, just use the `.json()` method on the `Response` object to get the decoded result:

```
response = requests.get('http://example.com/')
decoded_result = response.json()
```

However, this does not fail gracefully; it will raise a `JSONDecodeError` if the response object is not JSON-parseable.

You may wish to first check the content MIME type, for more graceful error handling:

```
if 'application/json' in response.headers['Content-Type']:
    decoded_result = response.json()
else:
    non_json_result = response.data
```

ETL from web API's with modules json and requests

First, import modules and set connection strings. If you need parameters, you can either put them directly in the URL string (an API in this case) or build them as a dict and pass them to the params argument.

```
import requests
import json

params = {'id': 'blahblah', 'output': 'json'} # You could use
```

```
https://www.somesite.com/api/query?id=blahblah&output=json directly.
API = 'https://www.somesite.com/api/query'
APIcred = 'username','password'
```

Requests handles HTTPBasicAuth and HTTPDigestAuth automatically. This example API will return a JSON string. Make the GET request and capture the output. Raise an error for bad HTTP status.

```
r = requests.get(API, params = params, auth = APIcred)
r.raise_for_status()
#print(r.status) # Optionally print HTTP status code
```

Convert string of JSON to python object you can work with. JSON looks visually similar to like a python dict, but there are significant differences in nulls, true/false, etc.

```
r_dict = json.loads(r.text)
print(r_dict)
```

Imagine that the output you just printed comes from a multi-line, multi-column database and is difficult to read:

> {'row': [{'Country': 'United States', 'pid': 'cc12608f-4591-46d7-b8fe-6222e4cde074', 'Status': '', 'FormerLastName': '', 'Degree': 'Business Administration'}, {'Country': 'Britain', 'pid': 'c9f2c6f7-f736-49d3-8adf-fd8d533bbd58', 'Status': '', 'FormerLastName': '', 'Degree': 'General Management'}]}

You can print() a more human-readable version with json.dumps(). The below line encodes the python object to a string of JSON with tabs and prints it.

```
print(json.dumps(r_dict, indent = 4))
```

Output:

```
{
    "row": [
        {
            "Country": "United States",
            "pid": "cc12608f-4591-46d7-b8fe-6222e4cde074",
            "Status": "",
            "FormerLastName": "",
            "Degree": "Business Administration"
        },
        {
            "Country": "Britain",
            "pid": "c9f2c6f7-f736-49d3-8adf-fd8d533bbd58",
            "Status": "",
            "FormerLastName": "",
            "Degree": "General Management"
        }
    ]
}
```

You can access nested elements in a dict like this:

```
print(some_dict['BuildingA']['Room12'])
```

But our sample has an arbitrary number of objects in an array, which is itself nested as the value of a key! These can be accessed with a row number, starting with 0.

Let's change one of our 'Country' values from 'Britain' to 'Albania':

```
r_dict['row'][1]['Country'] = 'Albania'
```

Now let's send this data to another API. Requests can accept a dict directly to the json argument, as opposed to encoding a string with json.dumps().

```
r = requests.post('https://www.somesite.com/" + 'api/carrots', json = r_dict, auth = APIcred)
r.raise_for_status()
```

Read Sending and receiving JSON online: https://riptutorial.com/python-requests/topic/3099/sending-and-receiving-json

Chapter 6: Using requests behind a proxy

Examples

Setting proxy in Python code

If your code is running behind a proxy and you know the end point, you can set this information in your code.

`requests` accepts a `proxies` parameter. This should be a dictionary that maps protocol to the proxy URL.

```
proxies = {
  'http': 'http://proxy.example.com:8080',
  'https': 'http://secureproxy.example.com:8090',
}
```

Notice that in the dictionary we have defined the proxy URL for two separate protocols: HTTP and HTTPS. Each maps to an individual URL and port. This does not mean that the two can't be the same, though. This is also acceptable:

```
proxies = {
  'http': 'http://secureproxy.example.com:8090',
  'https': 'http://secureproxy.example.com:8090',
}
```

Once your dictionary is defined, you pass it as a parameter.

```
requests.get('http://example.org', proxies=proxies)
```

Using proxy environment variables

`requests` uses specific environment variables automatically for proxy detection.

- `HTTP_PROXY` will define the proxy URL to use for HTTP connections
- `HTTPS_PROXY` will define the proxy URL to use for HTTPS connections

Once these environment variables are set, the Python code does not need to pass anything to the `proxies` parameter.

```
requests.get('http://example.com')
```

Read Using requests behind a proxy online: https://riptutorial.com/python-requests/topic/5933/using-requests-behind-a-proxy

Credits

S. No	Chapters	Contributors
1	Getting started with python-requests	Andy, Batsu, Burhan Khalid, Community, dansyuqri, icesin, k-nut, Martijn Pieters, Meysam, mickeyandkaka, mphuie, OJFord, supersam654
2	Automating login using Requests over Single Sign On	lordzuko
3	Django Framework	Burhan Khalid
4	Files	ReverseCold
5	Sending and receiving JSON	bendodge, Martijn Pieters, OJFord, Olli, Stan
6	Using requests behind a proxy	Andy