Learning Objectives

- **Get and parse content from an API with the `Requests` library**
- **Raise an exception if the status code indicates an error**
- **Use `Requests` to `PUT` and `POST` information to an API**
- **Add a header for authenticating against an API**
- **Use the `params` argument to simplify searching with URL parameters**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

Making Requests

Intro

Requests is the "de-facto" third-party HTTP library for Python. While Python does include its own urllib.request library, for making HTTP requests, even this module's documentation recommends using Requests for higher-level HTTP tasks.

To install Requests, just use `pip`:

```
pip3 install requests
```

Although, don't be surprised if it's already installed as lots of third-party packages rely on Requests. For example, in the Blango environment Requests is already installed because it's a dependency of Django-Allauth.

Once it's installed, let's see how to use it.

Making Requests

Using Requests is really simple (there's a reason it's so popular!). Just import the `requests` module and call the function with the name of the HTTP request method you want to use. Start the Python shell by entering the command below into the terminal.

```
python3 manage.py shell
```

▼ **JSONPlaceHolder**
In these examples we're requesting from JSONPlaceHolder, a free service that returns mock JSON objects for testing. As such you can follow along with the examples in a Python terminal, using the exact URLs if you like.

For example, here's how to `GET` from a URL:

```
In [1]: import requests

In [2]: response =
        requests.get("https://jsonplaceholder.typicode.com/posts
        /1")
```

`response` is an instance of Request's `Response` object. It has a number of useful methods/attributes that you can use to find out about what the server returned.

- `status_code`: contains the status code of the response:

```
In [3]: response.status_code
Out[3]: 200
```

- `text`: The response body in text form.

```
In [4]: response.text
Out[4]: '{\n  "userId": 1,\n  "id": 1,\n  "title": "sunt aut
        facere repellat provident occaecati excepturi optio
        reprehenderit",\n  "body": "quia et suscipit\\nsuscipit
        recusandae consequuntur expedita et cum\\nreprehenderit
        molestiae ut ut quas totam\\nnostrum rerum est autem
        sunt rem eveniet architecto"\n}'
```

- `content`: The response body in binary format.

```
In [5]: response.content
Out[5]: b'{\n  "userId": 1,\n  "id": 1,\n  "title": "sunt aut
        facere repellat provident occaecati excepturi optio
        reprehenderit",\n  "body": "quia et suscipit\\nsuscipit
        recusandae consequuntur expedita et cum\\nreprehenderit
        molestiae ut ut quas totam\\nnostrum rerum est autem
        sunt rem eveniet architecto"\n}'
```

While it looks similar to `text` notice that it is preceded by a `b` indicating it is binary data and thus has no encoding applied. Use `content` for binary data (like images) and `text` otherwise.

- `json()`: This method parses the body as JSON into a Python object.

```
In [6]: response.json()
Out[6]: {'body': 'quia et suscipit\nsuscipit recusandae
        consequuntur expedita et cum\nreprehenderit molestiae ut
        ut quas totam\nnostrum rerum est autem sunt rem eveniet
        architecto', 'id': 1, 'title': 'sunt aut facere repellat
        provident occaecati excepturi optio reprehenderit',
        'userId': 1}
```

- `headers`: A dictionary containing the response's HTTP headers:

```
In [7]: response.headers
Out[7]: {'Date': 'Fri, 17 Sep 2021 01:10:42 GMT', 'Content-
        Type': 'application/json; charset=utf-8', 'Transfer-
        Encoding': 'chunked', 'Connection': 'keep-alive', 'x-
        powered-by': 'Express', 'x-ratelimit-limit': '1000', 'x-
        ratelimit-remaining': '999', 'x-ratelimit-reset':
        '1631657191', 'vary': 'Origin, Accept-Encoding',
        'access-control-allow-credentials': 'true', 'cache-
        control': 'max-age=43200',
# ... and more
}
```

This is actually a case-insensitive dictionary where the case of the keys doesn't matter:

```
In [8]: response.headers["CoNtEnt-TyPe"]
Out[8]: 'application/json; charset=utf-8'
```

- `raise_for_status()`: This is a useful helper method that will raise an exception if the response `status_code` indicates an error. If we call it for our successful response, it does nothing:

```
In [9]: response.raise_for_status()
```

However, let's try it with a response that has a `404` status.

```
In [10]: response =
         requests.get("https://jsonplaceholder.typicode.com/inval
         idurl")



In [11]: response.raise_for_status()
---------------------------------------------------------------
         -----------
HTTPError                                 Traceback (most recent
         call last)
<ipython-input-4-98371c55c61f> in <module>()
----> 1 response.raise_for_status()

/usr/lib/python3/dist-packages/requests/models.py in
         raise_for_status(self)
    933
    934         if http_error_msg:
--> 935             raise HTTPError(http_error_msg,
         response=self)
    936
    937     def close(self):

HTTPError: 404 Client Error: Not Found for url:
         https://jsonplaceholder.typicode.com/invalidurl
```

Notice the exception is of type `requests.exceptions.HTTPError`.

You would use `raise_for_status()` in place of having to write your own exception handling to abort the execution of your code when dealing with bad responses. For example, without using `raise_for_status()` you might have to write some code like this:

Note: This is an example, *do not* enter this into the shell.

```
response = request.get(url)

if response.status_code > 399:
    raise MyHttpError()  # assume you've written a class called
        MyHttpError()

do_something_with_response(response)
```

Instead you could shortcut it like this:

```
response = request.get(url)

response.raise_for_status()

do_something_with_response(response)
```

There are other methods and attributes on the `Response` object, but these are the only ones that will be useful to us. If you're interested in seeing how to stream responses or detect redirects, check the Requests and Response Objects documentation.

Other Requests

Other Requests

You might have already guessed how to make other types of requests. For example, to `DELETE` something:

Note: The following code examples are to be read and not entered into the IDE.

```
>>> response =
        requests.delete("https://jsonplaceholder.typicode.com/po
        sts/1")
>>> response.status_code
200
```

▼ **Persistent changes**
Note that with this placeholder API, changes to data are not persisted and so this only pretends to delete on the server. But the response mimics an actual successful deletion.

To `POST` or `PUT` data, we can include data with the `data` argument to the method. If we're `POST`ing to a regular HTML form the data can just be supplied as a dictionary. For example, perhaps we want to fake a login to our Django server:

```
requests.post("http://127.0.0.1:8000/accounts/login/", data=
        {"username": "ben@example.com", "password": "password"})
```

▼ **CSRF token**
Note that this example won't work as you'd need to also pass the CSRF token.

However if we want to send JSON data, we can either encode it ourselves:

```
import json

post_data = { "id": 1, "title": "New Title", "body": "Updated
        Body", "userId": 1}


resp =
        requests.put("https://jsonplaceholder.typicode.com/posts
        /1", data=json.dumps(post_data))
```

Or, we can pass it as the `json` argument, and Requests will encode it for us:

```
resp =
        requests.put("https://jsonplaceholder.typicode.com/posts
        /1", json=post_data)
```

When using the `json` argument, Requests automatically sets the `Content-Type` HTTP header to `application/json`. When using the `data` argument, if your server requires it, it can be set manually, using the `headers` argument. This takes a dictionary of HTTP headers to send with the request. So, to set the appropriate `Content-Type` header:

```
resp =
        requests.put("https://jsonplaceholder.typicode.com/posts
        /1", data=json.dumps(post_data), headers={"Content-
        Type": "application/json"})
```

If using `json` automatically encodes as JSON *and* sets the right `Content-Type` header, why would you use the `data` argument? Requests uses the built-in Python JSON encoder, which can't encode `datetime` objects or custom classes/types. If you need to use your own JSONEncoder, then you'll have to encode the data with it first and then pass it to `data`.

Now let's look a little more at authentication and headers.

Headers and Authorization

We just saw how easy it is to add headers to a request using the `headers` argument. There's not much more to say on the topic, except for a demo on how to authorize requests. The process is similar to what was used when writing tests using the `RequestsClient`: make a request to an endpoint that provides a token, then use it in the `Authorization` header.

Try It Out

Here's a short script that you can create to demonstrate and test authentication with a token. It makes a request against the Blango API to get a list of `Post`s. Then authenticates and makes another request, getting

the list of `Posts` as an authenticated user.

Before you run this script you'll need to have your Django Dev Server running.

Switch back to the tab entitled `requests_test.py`, and enter the code below. Be sure to put **your** credentials in the `EMAIL_ADDRESS` and `PASSWORD` variables.

```
import requests

# put your real credentials here
EMAIL_ADDRESS = "ben@example.com"
PASSWORD = "password"
BASE_URL = "http://localhost:8000/"

anon_post_resp = requests.get(BASE_URL + "api/v1/posts/")
anon_post_resp.raise_for_status()

anon_post_count = anon_post_resp.json()["count"]
print(f"Anon users have {anon_post_count} post{'' if
        anon_post_count == 1 else 's'}")

auth_resp = requests.post(
    BASE_URL + "api/v1/token-auth/",
    json={"username": EMAIL_ADDRESS, "password": PASSWORD},
)
auth_resp.raise_for_status()
token = auth_resp.json()["token"]

# Use the token in a request
authed_post_resp = requests.get(
    BASE_URL + "api/v1/posts/", headers={"Authorization":
        f"Token {token}"}
)
authed_post_count = authed_post_resp.json()["count"]

print(
    f"Authenticated user has {authed_post_count} post{'' if
        authed_post_count == 1 else 's'}"
)

# Since requests doesn't remember headers between requests, this
        next request is unauthenticated again
anon_post_resp = requests.get(BASE_URL + "api/v1/posts/")
```

Notice we are authenticating DRF's built in token system. We could just as easily use JWTs by changing the URL that's being used for authorization and the header prefix (from `Token` to `JWT`). Run the script and see if there's

any difference in the number of `Posts` that the logged in user can see compared to the anonymous user:

```
Anon users have 6 posts
Authenticated user has 11 posts
```

There may or may not be difference in these two lists, but if each list has the same `count` it's not a sign that something's gone wrong.

We'll finish our quick look at Requests by discussing how to use URL (query) parameters.

URL Parameters

URL Parameters

When using Requests to make requests that include query parameters, you shouldn't try to build the URL yourself. This is because you can use the `params` argument which will automatically take care of escaping for you.

For example, in our `Posts` API, we can search for `Posts` based on their content. If we wanted to search for `Posts` by the author with PK `1`, whose content contains `Python`, we might try to make a request like this:

```
posts = requests.get("http://127.0.0.1:8000/api/v1/posts/?
        author=1&content=Python")
```

Our API would receive the parameters `author` with value `1` and `content` with value `Python`.

Now if we wanted to search for `Posts` containing the content `Python & Django`, maybe we'd do this:

```
posts = requests.get("http://127.0.0.1:8000/api/v1/posts/?
        author=1&content=Python & Django")
```

▼ **Search logic**
Note that we have not implemented complicated search logic for our API, so this would mean we're searching for the literal phrase `Python & Django` in the content, rather than `Posts` that contain `Python` somewhere in the content, and also `Django` somewhere in the content.

Doing this would mean our API receives three parameters:

- `author`, with value `1`
- `content` with value `Python` (note the trailing space)
- and `Django` (with leading space), which has no value

As well as just `&` we'd also have to escape all other special characters that affect URLs. This is why we should just let Requests handle it by passing a `params` dictionary. The proper way to make our request would be like this:

```
posts = requests.get("http://127.0.0.1:8000/api/v1/posts/",
        params={"author": "1", "content": "Python & Django"})
```

We can also provide a `list` as a value in the `params` dictionary. For example, to search by multiple `tags`, we need to specify them all in the URL, for example: `/api/v1/posts/?tags=1&tags=5`.

This request is made through Requests like this:

```
posts = requests.get("http://127.0.0.1:8000/api/v1/posts/",
        params={"tags": ["1", "5"]})
```

That's all we're going to cover for the Requests module. Next we're going to look at how to use Requests to interact with a real API, and how to model its data in Django.

Pushing to GitHub

Pushing to GitHub

No changes were made to the `Blango` project. You used the Python shell to work with requests. The `requests_test.py` file is not a part of `Blango`. There is nothing to push to GitHub.