

HTTP GET requests and JSON

What is HTTP?

- Hypertext Transfer Protocol (HTTP) is a set of rules that allow web browsers and web servers to communicate.
- HTTP methods include:
 - GET - retrieve a resource
 - POST - submit
 - PUT - replace
 - DELETE
- These methods can correspond to CRUD operations for a data API (more on this later)

Imports for this lab

```
import requests
import json
import os
from bson import ObjectId, json_util
```

HTTP requests

Use python requests library to make HTTP requests.

```
r = requests.get('https://api.github.com/events')
print(r.status_code)
# print(r.text)

# Open file and use Shift + Alt + F to prettify
dir_path = os.path.dirname(os.path.realpath(__file__))
with open(f'{dir_path}/events.json', 'w', encoding='utf-8') as f:
    f.write(r.text)
```

Inspecting JSON

- At top level, this JSON data is an array of objects.
- Each JSON object has one or more field(s).
- A JSON field consists of a key and a value.

```
[  
  {  
    "id": "46130780958",  
    "type": "PushEvent",  
    "actor": {  
      "id": 66974840,  
      "login": "levanydze",  
      "display_login": "levanydze",  
      [...]
  }
]
```

Loading JSON and accessing data

- `json.loads` will convert JSON text into a python object
- below, we get the first event, then the actor and login.
- event and actor are python dicts, login is a str
- we can combine statements

```
events = json.loads(r.text)
event = events[0]
actor = event.get('actor')
login = actor.get('login')
print(login)

# print(events[0].get('actor').get('login'))
```

Dumping JSON

```
actor = events[0].get('actor')  
json_str = json.dumps(actor)  
print(json_str)
```

ObjectIds and JSON

- recall that MongoDB uses BSON ObjectIds
- what happens when you run the code below?

```
my_object = {  
    'id': ObjectId(),  
    'name': 'something'  
}  
json.dumps(my_object)
```


Serialization

- Serialization is the process of converting data into a format that can be stored or transmitted
- When we dump python objects to JSON strings, we are serializing.
- We have options for serializing an objectId
 - use json_util
 - use strings

Using json_util

- json_util.dumps will serialize ObjectIds
- you can deserialize with json.loads and you will get a plain python dict.
- if you deserialize with json_util.loads, you will get an ObjectId

```
json_str = json_util.dumps(my_object)
print(json_str)
# {"id": {"$oid": "679e09f4544c45b2ddc8b381"}, "name": "something"}

o = json_util.loads(json_str)
print(o)
# {'id': ObjectId('67b4aa6408aea7a1ac2ad8ab'), 'name': 'something'}
```

Using plain strings

- Assign object a string id
- A benefit of this approach is that your API consumer can just use regular JSON, and doesn't have to worry about BSON or ObjectIds.

```
my_object = {
    'id': str(ObjectId()),
    'name': 'something'
}
json_str = json.dumps(my_object)
print(json_str)
# {"id": "679e0c3e4a9bc6016ac3e81f", "name": "something"}

o = json.loads(json_str)
print(o.get('id'))
# 679e0c3e4a9bc6016ac3e81f
```