Data Collection

The first step of most data science pipelines, as you may imagine, is to get some data. Data that you typically use comes from many different sources. If you're lucky, someone may hand directly had you a file, such as a CSV. Or sometimes you'll need to issue a database query to collect the relevant data . But in this lecture, we'll talk about collecting data from two main sources: 1) querying an API (the majority of which are web-based, these days); and 2) scraping data from a web page.

Collecting data from web-based sources

The vast majority of automated data queries you will run will use HTTP requests (it's become the dominant protocol for much more than just querying web pages)

```
import requests
response = requests.get("https://fmi.chnu.edu.ua/")

print("Status Code:", response.status_code)
print("Headers:", response.headers)
```

```
Status Code: 200
Headers: {'Date': 'Sun, 18 Aug 2024 13:47:40 GMT', 'Content-Type':
'text/html; charset=utf-8', 'Transfer-Encoding': 'chunked',
'Connection': 'keep-alive', 'vary': 'Accept-Encoding', 'content-
security-policy': "default-src 'self' data: 'unsafe-eval' 'unsafe-
inline' *.gstatic.com *.googleapis.com *.googletagmanager.com
*.addtoany.com *.youtube-nocookie.com *.google.com *.google-
analytics.com *.ytimg.com *.facebook.com forms.gle *.chnu.edu.ua
madmagz.com", 'x-frame-options': 'SAMEORIGIN, SAMEORIGIN', 'x-content-
type-options': 'nosniff', 'strict-transport-security': 'max-
age=31536000', 'referrer-policy': 'no-referrer', 'permissions-policy':
'accelerometer=(), camera=(), geolocation=*, gyroscope=(),
magnetometer=(), microphone=(), payment=(), usb=()', 'CF-Cache-
Status': 'DYNAMIC', 'Report-To':
'{"endpoints":[{"url":"https:\\/\\/a.nel.cloudflare.com\\/report\\/v4?
s=jzsoc%2F9fmYB%2BNLhbPj7MGfIeQ0snoYVSMd7GURUDaxyfE6FjjTLRe
%2BUn4%2BIEspNxELuFs
%2FjkJ4H2FJBZlWAWcLqJjIeQ9bUJi2X3RsrlgmqbmvwDFego09CnWD0oUPm4occ
%3D"}],"group":"cf-nel","max_age":604800}', 'NEL':
'{"success_fraction":0,"report_to":"cf-nel","max_age":604800}',
'Server': 'cloudflare', 'CF-RAY': '8b525b6608115d55-FRA', 'Content-
Encoding': 'gzip', 'alt-svc': 'h3=":443"; ma=86400'}
```

```
print(response.text[:480])
```

```
<!DOCTYPE html>
<html lang="uk" prefix="og: https://ogp.me/ns#">
```

```
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1">
        <title>&#x413;&#x43E;&#x43B;&#x43E;&#x432;&#x43D;&#x430; -
&#x424;&#x430;&#x43A;&#x443;&#x43B;&#x44C;&#x442;&#x435;&#x442;
&#x43C;&#x430;&#x442;&#x435;&#x43C;&#x430;&#x442;&#x438;&#x43A;&#x438;
&#x442;&#x430;
&#x456;&#x43D;&#x444;&#x43E;&#x440;&#x43C;&#x430;&#x442;&#x438;&#x43A;
&#x438;</title
```

You've seen URLs like these: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=9&cad=rja&uact=8… The weird statements after the url are parameters, you would provide them using the requests library like this:

```
params = {"sa":"t", "rct":"j", "q":"", "esrc":"s",
"source":"web", "cd":"9", "cad":"rja", "uact":"8"}
response = requests.get("http://www.google.com/url", params=params)
```

HTTP GET is the most common method, but there are also PUT, POST, DELETE methods that change some state on the server

RESTful APIs

If you move beyond just querying web pages to web APIs, you'll most likely encounter REST APIs (Representational State Transfer) REST is more a design architecture, but a few key points:

1. Uses standard HTTP interface and methods (GET, PUT, POST, DELETE)
2. Stateless – the server doesn't remember what you were doing Rule of thumb: if you're sending your account key along with each API call, you're probably using a REST API

You query a REST API similar to standard HTTP requests, but will almost always need to include parameters

Get your own access token at https://github.com/settings/tokens/new GitHub API uses GET/PUT/DELETE to let you query or update elements in your GitHub account automatically Example of REST: server doesn't remember your last queries, for instance you always need to include your access token if using it this way

```
token = ""
headers = {'Authorization': 'token '+token}
response = requests.get("https://api.github.com/user",
headers=headers)
print(response.content)
```

Data formats

The three most common formats:

1. CSV (comma separate value) files
2. JSON (Javascript object notation) files and strings
3. HTML/XML (hypertext markup language / extensible markup language) files and strings

CSV files

Refers to any delimited text file (not always separated by commas)

If values themselves contain commas, you can enclose them in quotes (our registrar apparently always does this, just to be safe)

```
import pandas as pd

dataframe = pd.read_csv("resources/example.csv", delimiter=',',
quotechar='"')
```

JSON files / string

JSON originated as a way of encapsulating Javascript objects A number of different data types can be represented

- Number: 1.0 (always assumed to be floating point)
- String: "string"
- Boolean: true or false
- List (Array): [item1, item2, item3,...]
- Dictionary (Object in Javascript): {"key":value}
- Lists and Dictionaries can be embedded within each other: [{"key":[value1, [value2, value3]]}]

Example JSON data

JSON from Github API

```
{
'login': 'pvirtue',
'id': 5945661,
'node_id': 'MDQ6VXNlcjU5NDU2NjE=',
'avatar_url': 'https://avatars.githubusercontent.com/u/5945661?v=4',
...
```

Parsing JSON in Python

Built-in library to read/write Python objects from/to JSON files

```
import json
# load json from a REST API call

headers = {'Authorization': 'token ' + token}
```

```
response = requests.get("https://api.github.com/user",
headers=headers)
data = json.loads(response.content)

#json.load(file) # load json from file
#json.dumps(obj) # return json string
#json.dump(obj, file) # write json to file
```

XML / HTML files

The main format for the web (though XML seems to be loosing a bit of popularity to JSON for use in APIs / file formats)

```
<tag attribute="value">
    <subtag>
        Some content for the subtag
    </subtag>
    <openclosetag attribute="value2”/>
</tag>
```

XML files contain hiearchical content delineated by tags HTML is syntactically like XML but not as strict (e.g., open tags are not always closed)

Parsing XML/HTML in Python

There are a number of XML/HTML parsers for Python, but a nice one for data science is the BeautifulSoup library (specifically focused on getting data out of XML/HTML files)

```
# get all the links within the webpage
from bs4 import BeautifulSoup
import requests

response = requests.get("http://www.cs.utsa.edu/lectures")

root = BeautifulSoup(response.content)
root.find("table").find("tbody").findAll("a")
```

Regular expressions

Once you have loaded data (or if you need to build a parser to load some other data format), you will often need to search for specific elements within the data.

E.g., find the first occurrence of the string “data science”

```
import re
text = "This course will introduce the basics of data science"
match = re.search(r"data science", text)
print(match.start())
```

```
# Regular expressions in Python

match = re.match(r"data science", text) # check if start of text
matches
match = re.search(r"data science", text) # find first match or None

all_matches = re.findall(r"data science", text) # return all matches

# You can also use “compiled” version of regular expressions
# regex = re.compile(r"data science")
# regex.match(text, [startpos, [endpos]])
# regex.search(...)
# regex.finditer(...)
# regex.findall(...)
```

Matching multiple potential characters

The real power of regular expressions comes in the ability to match multiple possible sequence of characters. Special characters in regular expressions: `.^$*+?{}\[]|()` (if you want to match these characters exactly, you need to escape them: `\$`)

Match sets of characters:

- Match the character ‘a’: `a`
- Match the character ‘a’, ‘b’, or ‘c’: `[abc]`
- Many any character except ‘a’, ‘b’, or ‘c’: `[^abc]`
- Match any digit: `\d` (the same as `[0-9]`)
- Match any alpha-numeric: `\w` (the same as `[a-zA-z0-9_]`)
- Match whitespace: `\s` (the same as `[ \t\n\r\f\v]`)
- Match any character: . (including newline with re.DOTALL)

Can match one or more instances of a character (or set of characters)

Some common modifiers:

- Match character ‘a’ exactly once: `a`
- Match character ‘a’ zero or one time: `a?`
- Match character ‘a’ zero or more times: `a*`
- Match character ‘a’ one or more times: `a+`
- Match character ‘a’ exactly n times: `a{n}`

Can combine these with multiple character matching:

- Match all instances of “`<something>` science” where `<something>` is an alphanumeric string with at least one character
- `\w+\s+science`

Grouping

We often want to obtain more information that just whether we found a match or not (for instance, we may want to know what text matched)

Grouping: enclose portions of the regular expression in parentheses to "remember" these portions of the match `(\w+)\s([Ss]cience)`

```
match = re.search(r"(\w+)\s([Ss]cience)", text)
print(match.start(), match.groups())
# Why the 'r' before the string? Avoids need to double escape strings
```

Substitutions

Regular expressions provide a power mechanism for replacing some text with other text

```
better_text = re.sub(r"data science", r"schmada science", text)

# To include text that was remembered in the matching using groups,
use the escaped sequences
# \1, \2, … in the substitution text
better_text = re.sub(r"(\w+)\s([Ss])cience", r"\1 \2hmience", text)
```

Ordering and greedy matching

There is an order of operations in regular expressions. `abc|def` matches the strings "abc" or "def", not "ab(c or d)ef". You can get around this using parenthesis e.g. `a(bc|de)f`. This also creates a group, use `a(?:bc|de)f` if you don't want to capture it. By default, regular expressions try to capture as much text as possible (greedy matching). `<(.*)>` applied to `<a>text</a>` will match the entire expression. If you want to capture the least amount of text possible use `<(.*?)>` this will just match the `<a>` term.