Data Loading

Python and R for Data Science

Data Science and Management

Preliminaries

Data Science

In data science, we have few key steps:

1. Get the dataset
2. Load the dataset
3. Clean the dataset
4. Process and analyze the dataset
5. Visualize the dataset

Throughout the course, we will cover these steps and refine them.

Nonetheless, the starting point is... **obtain the data**.

Many faces of the data

About the data, we need to understand:

- *where it is stored*:
    - ***locally*** (e.g., our disk)
    - ***remotely*** (e.g., website)
- *how it is stored*: its data structure, i.e., its data ***format***

Data Collection

Keep everything local?

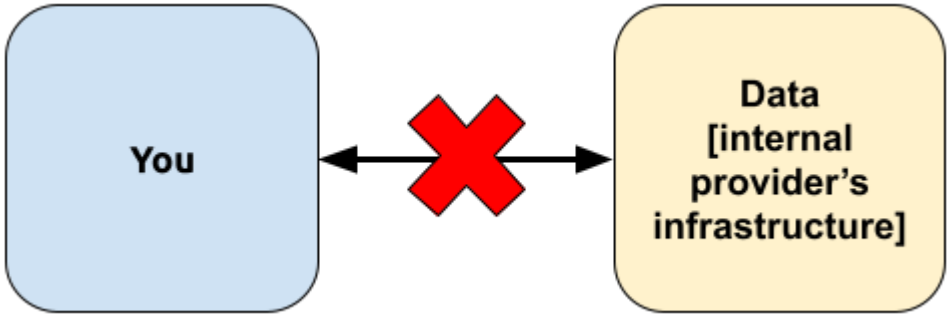
In principle, we may want to have the entire dataset on our local machine. For instance, by manually downloading the dataset from the web.

However, this is not always possible:

- *the dataset is too large*: we can only locally store chunks of data
- *the dataset is not fully available*: several web services do not allow us to obtain the entire dataset for different (good or bad) reasons. For instance:
    - A service is selling the access to the data and wants to give you limited access to it.
    - Privacy concerns (e.g., health sector) that require to track exactly the data that you request to collect.
    - Give you the entire dataset would generate excessive network traffic.
- *the dataset is live*: it has frequent updates.

Remote access to a dataset?

A *data provider* may store the dataset in quite different and undocumented ways.
Most likely, there will be for you **no direct access** to the dataset or to the system handling the data.

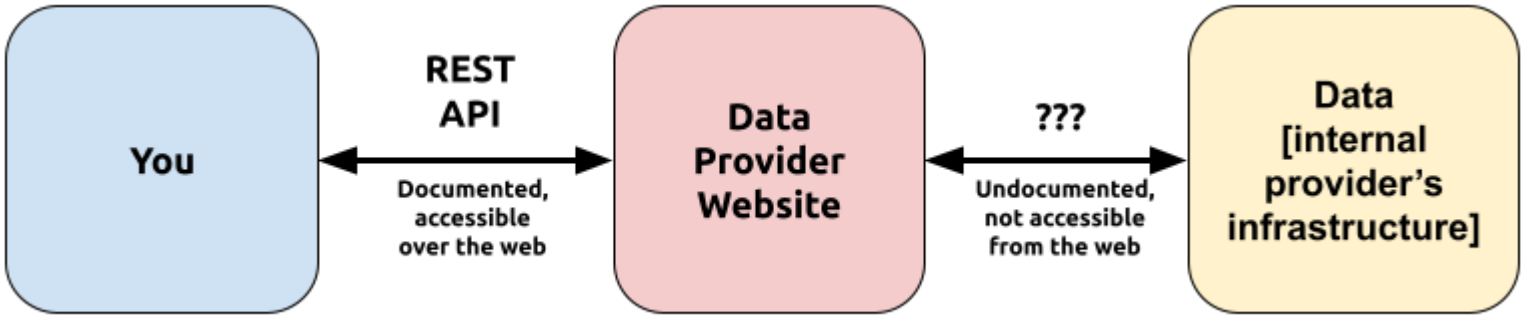


Indeed, the data provider:

- wants to control how you access the data (*pay as you call*)
- does not want to give you insights about its internal infrastructure:
    - it may lead to security issues
    - it may lead to data leaks
    - it wants to be free to change how it works over time

Then, how to access the remote data? **REST API**

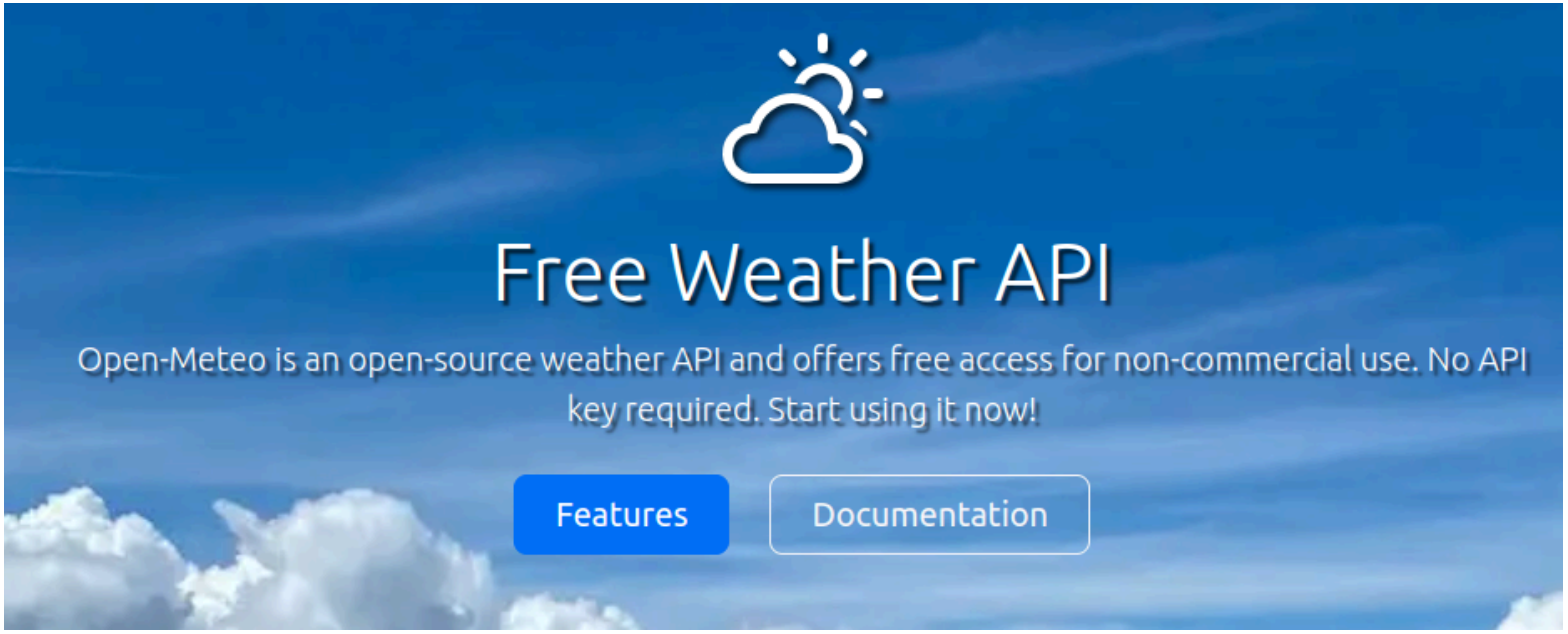
For these reasons, datasets are often exposed to the external world through a *standard remote interface* called ***REST API*** (or, RESTful API):



A REST API:

- is simple to implement (provider) and use (users)
- is typically reachable from the internet via HTTPS (i.e., web)
- can easily integrate authentication
- is often documented by the data provider
- can be fine-grained: users can retrieve exactly the needed bit of data
- can be inefficient: there are better solutions that, however, are less widespread and more complex

Example: open-meteo.com



The free access is limited to a subset of the data!
Data has a huge money value and most of the time you will have to pay for it!

Example: open-meteo.com (cont'd)

API Documentation

The API endpoint `/v1/forecast` accepts a geographical coordinate, a list of weather variables and responds with a JSON hourly weather forecast for 7 days. Time always starts at 0:00 today and contains 168 hours. If `&forecast_days=16` is set, up to 16 days of forecast can be returned. All URL parameters are listed below:

Parameter	Format	Required	Default	Description
latitude, longitude	Floating point	Yes		Geographical WGS84 coordinates of the location. Multiple coordinates can be comma separated. E.g. `&latitude=52.52,48.85&longitude=13.41,2.35`. To return data for multiple locations the JSON output changes to a list of structures. CSV and XLSX formats add a column `location_id`.
elevation	Floating point	No		The elevation used for statistical downscaling. Per default, a 90 meter digital elevation model is used. You can manually set the elevation to correctly match mountain peaks. If `&elevation=nan` is specified, downscaling will be disabled and the API uses the average grid-cell height. For multiple locations, elevation can also be comma separated.
hourly	String array	No		A list of weather variables which should be returned. Values can be comma separated, or multiple `&hourly=` parameter in the URL can be used.
daily	String array	No		A list of daily weather variable aggregations which should be returned. Values can be comma separated, or multiple `&daily=` parameter in the URL can be used. If daily weather variables are specified, parameter `timezone` is required.

Full documentation: **https://open-meteo.com/en/docs**

Example: open-meteo.com (cont'd)

Data can be accessed on the web via the endpoint: https://api.open-meteo.com/v1/forecast

For instance, after checking the documentation, we can build the following request:

https://api.open-meteo.com/v1/forecast?latitude=41.8967&longitude=12.4822&hourly=temperature_2m

which returns the last temperature measurements in Rome. You can open this URL with your browser:


api.open-meteo.com/v1/forecast?latitude=41.8967&longitude=12.4822&hourly=temperature_2m
Pretty-print
{
"latitude": 41.9,
"longitude": 12.48,
"generationtime_ms": 0.0550746917724609,
"utc_offset_seconds": 0,
"timezone": "GMT",
"timezone_abbreviation": "GMT",
"elevation": 31,
"hourly_units": {
"time": "iso8601",
"temperature_2m": "°C"
},
"hourly": {
"time": [
"2024-08-02T00:00",
"2024-08-02T01:00",
"2024-08-02T02:00",
"2024-08-02T03:00",
"2024-08-02T04:00",
"2024-08-02T05:00",
"2024-08-02T06:00",

Example: open-meteo.com (cont'd)

If we look closely at the response, we can see the data that we care for:


```
   "temperature_2m": [28, 28, 27.4, 25.9, 25.7, 26.9, 27.8, 29.2, 31.5, 32.7, 34.7, 34.7, 34.8, 34.8, 33.7, 32.6, 32.2, 31.6, 30.4, 29.8, 29.2, 28.8,
28.4, 28.4, 28.4, 28.2, 27.3, 27.1, 26.9, 27, 27.9, 28.9, 30.5, 31.8, 33, 34.7, 36.1, 36, 35.9, 35.8, 32.6, 32.7, 31.8, 30.7, 30.2, 30.3, 29.4, 28.2,
27.3, 26.7, 26.2, 25.9, 25.8, 25.9, 27.2, 28.6, 30.6, 32.5, 34.7, 36.3, 35.5, 35.2, 35, 34.5, 33.5, 32.4, 31.6, 30.3, 29.4, 29.1, 28.5, 28.1, 27.9,
27.8, 27.8, 27.6, 27.3, 27.2, 27.7, 29, 30.5, 32.2, 33.5, 34.5, 34.9, 34.5, 34.9, 34.5, 33.6, 32.8, 31.7, 30.7, 29.9, 29.3, 28.7, 28.4, 28.2, 27.9,
27.7, 27.5, 27.2, 27.1, 27.8, 29.2, 30.9, 32.7, 34.6, 36.2, 36.5, 35.9, 35.3, 34.1, 33.3, 32.7, 31.9, 29, 27.9, 26.9, 26.2, 25.6, 25.1, 24.6, 24.1,
24, 24.3, 25, 25.5, 27.5, 29.8, 31.9, 33.6, 35, 35.9, 35.8, 35.2, 34.4, 33.7, 32.9, 31.9, 30.6, 29.2, 27.9, 27, 26.4, 25.8, 25.2, 24.7, 24.6, 24.7,
25.1, 26.1, 28.1, 30.8, 32.9, 34.3, 35.3, 35.8, 35.8, 35.3, 34.7, 33.9, 32.9, 31.8, 30.4, 28.9, 27.5, 26.5, 25.6]
  }
}
```


These are the temperature measurements!

REST API in practice

Given a data provider, we need to:

- **[problem #1]** how to build the REST API requests: check the documentation!
- **[problem #2]** perform such **requests** in a fully automatic way:
    - while we could do them manually, it is not convenient
    - live data must be fetched periodically
- **[problem #3]** interpret the **responses**:
    - the response can be anything:
        - text file
        - image
        - raw data
    - we need to identify the proper **data format**
    - the data format is documented by the data provider
    - the data format will be standard to favor interoperability

REST API request

In REST API, a request:

- can have some ***request headers***: we may need to set some headers to, e.g., perform authentication

- is performed using a ***URL*** which combines:

    - ***request endpoint***, e.g., `https://api.open-meteo.com/v1/forecast`
    - ***request parameters***: e.g., `?latitude=41.8967&longitude=12.4822`
      The question mark `?` starts the paramters list, where each parameter is a key-value pair, such as `<parameter_name>=<parameter_value>`, or `<parameter_name>=<parameter_value1>,<parameter_value2>`, where key-value pairs are separated by the symbol `&`

- can be performed using two HTTP methods:

    - ***GET method***: the most common one (previous example). Any parameter will be embedded into the URL.
    - ***POST method***: more advanced, often used when we need to send some data that cannot be embedded through headers or request parameters

All these details will be written by the documentation.

How to automatically perform REST API requests?

- Data provider API Python package: some services, or the community, may offer a dedicated Python package. For instance, for the Open Meteo API, we could install with `pip` this package: https://pypi.org/project/open-meteo/
- We can use a popular and well-known Python package: `requests`
  It will work for any REST API. We can install it with `pip`:

In [4]:
```
! python3 -m pip install requests
```

```
Defaulting to user installation because normal site-packages is
not writeable
Requirement already satisfied: requests in /home/ercoppa/.loca
l/lib/python3.10/site-packages (2.31.0)
Requirement already satisfied: charset-normalizer<4,>=2 in /hom
e/ercoppa/.local/lib/python3.10/site-packages (from requests)
(3.1.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/lib/p
ython3/dist-packages (from requests) (2020.6.20)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/lib/p
ython3/dist-packages (from requests) (1.26.5)
Requirement already satisfied: idna<4,>=2.5 in /usr/lib/python
3/dist-packages (from requests) (3.3)
```

`requests`: make a GET request

First, we need to import it:

In [6]:
```
import requests
```

Then, we can make our first GET request:

In [9]:
```
url = 'https://api.open-meteo.com/v1/forecast?latitude=41.89&longitude=
response = requests.get(url)
```

Now, the response is stored in the `response` variable. Depending on the data format, we must parse it in a different way.

`requests`: request parameters

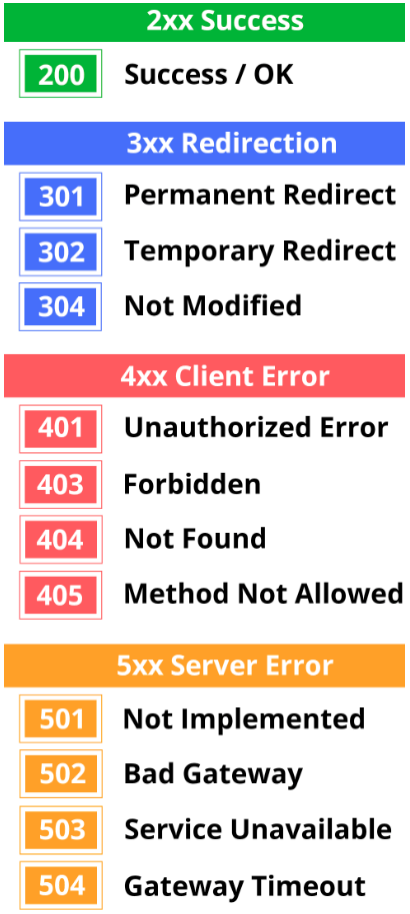
Instead of manually setting the list of parameters in the url, we can do:

In [26]:
```
endpoint = 'https://api.open-meteo.com/v1/forecast'
params = {'latitude': 41.89, 'longitude': 12.48, 'hourly': 'temperature
response = requests.get(endpoint, params=params)
```

which is way more readable and less error-prone!

Response status code

Each response come with a ***status code*** that indicates whether the request has been successfully completed:



[image credits]

`requests`: check response status code

In [11]:
```
url = 'https://api.open-meteo.com/v1/forecast?latitude=41.89&longitude=
response = requests.get(url)
print(f"Status code: {response.status_code}")
```

```
Status code: 200
```

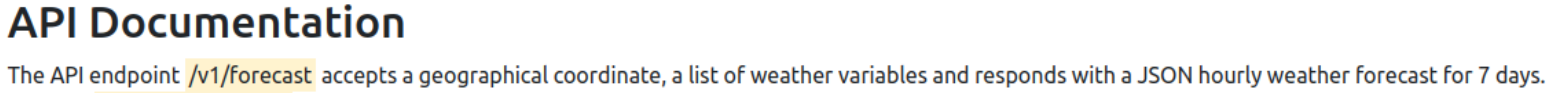
Our request was successful and we can now get the data our of it :)

`requests`: retrieve response data

`requests` allows us to retrieve the data in the following formats:

- `response.text`: textual format. Used only when the response data is a single value (e.g., a string, integer, etc.)
- `response.json()`: JSON format. The most common choice. Easy to parse (see later slides!).
- `response.content`: the response is an arbitrary format and we are getting the raw bytes. Need to carefully check the API documentation. It can be used when downloading files (e.g., an image, a zip file, etc.) from arbitrary websites.

For instance, in the case of Open Meteo API, the documentation reports that the response is in a JSON format:



`requests`: textual response

We treat the response data as a Python string:

In [15]:
```
response = requests.get('https://google.it')
print(response.text[:50]) # first 50 characters of google page
```

```
<!doctype html><html itemscope="" itemtype="http:/
```

`requests`: JSON response

A JSON is like a Python Dictionary:

In [28]:
```
url = 'https://api.open-meteo.com/v1/forecast?latitude=41.89&longitude=
response = requests.get(url).json()
print(f"Temperature: {response['hourly']['temperature_2m'][:10]}")
```

```
Temperature: [27.4, 27.8, 27.1, 25.8, 25.6, 26.7, 27.9, 29.4, 3
1.5, 32.7]
```

The structure of the dictionary, i.e., which key-value pairs are inside it, it expected to be documented by the data provider.

Hence, in our example, the documentation from Open Meteo is reporting that there is `hourly` key, whose associated value is a dictionary containing the key `temperature_2m`, whose associated value is a list of `float` values, i..e, our temperatures.

More details on the JSON format in later slides!

`requests`: raw response

When we want to download an arbitrary file over the web, we can get the file and save it to our local filesystem.

For instance:

In [21]:
```
# a pic from the web
url = "https://cdn.pixabay.com/photo/2023/11/14/20/08/woman-8388428_128
rawdata = requests.get(url).content # get the image
open('myimage.jpg', 'wb').write(rawdata) # saved the image to a file
```

Out[21]:
```
88584
```

After executing these lines of code, you have a local file `myimage.png`. You can open it with your image viewer.

`requests`: other features

This package is extremely powerful and make it easy to:

- perform a POST request:
  - use `requests.post(url, data=<our data>)`
- add headers to a request:
  - pass a dictionary with the key-value pairs
  - e.g., `request.get(url, headers={'User-Agent':'MyApp'})`

Look at its documentation for more details:

https://requests.readthedocs.io/en/latest/user/quickstart/

In future lectures, we may come back to `requests`.

Data Formats

Popular data formats

Datasets may come in different common formats:

- Textual: `.txt` file
- CSV: `.csv` file
- TSV: `.csv` file
- JSON: often retrieved via a REST API
- XSLX: `xlsx` file, Microsoft Excel format

Textual file (`.txt`)

The most intuitive one. Human-friendly but hard to parse when we have complex data inside it. Used in the real-world only when the data is indeed a simple text (e.g., a book). We can open the local file and fetch its content as a Python string.

Example: `myfile.txt`

```
Hello, LUISS!
Hello, World!
```

Remarks:

- Should we split it by lines? Most of the time makes sense to do it.
- No hint on how to split internally each line. Most of the time does not makes sense to do it.

Textual file (`.txt`): parsing

Since a textual file is just a string:

In [41]:
```
content = open('myfile.txt', 'r').read()
print(content)
```

```
Hello, LUISS!
Hello, World!
```

Since it is a string, we can manipulate it through the string functions and operators available in Python.
For instance, we may want to split by newline:

In [39]:
```
for i, line in enumerate(content.split("\n")):
    print(f"Line #{i}: {line}")
```

```
Line #0: Hello, LUISS!
Line #1: Hello, World!
```

Splitting the text file by line can done more quickly via:

In [43]:
```
print(open('myfile.txt', 'r').readlines())
```

```
['Hello, LUISS!\n', 'Hello, World!']
```

CSV file (`.csv`)

A *Comma-Separated Values* (**CSV**) file:

- is a text file format
- storing tabular data (numbers and text), where each line of the file typically represents one data record
- uses *commas* (`,`) or semicolon (`;`) to separate values and newlines to separate records
- quite easy to parse (even without dedicated support)
- the first line may contain the table headers, i.e., column names
- can be imported in Spreadsheet app (e.g., Excel)

Example: `myfile.csv`

```
Username,Identifier,First name,Last name
booker12,9012,Rachel,Booker
grey07,2070,Laura,Grey
```

Should be seen as:

Username	Identifier	First name	Last name
booker12	9012	Rachel	Booker
grey07	2070	Laura	Grey

CSV file (`.csv`): parsing

1. Manually: not hard but a bit raw

In [34]:
```
data = open('myfile.csv', 'r').read() # you have the prev example file!
for line in data.split('\n'):
    print(line.split(','))
```

```
['Username', 'Identifier', 'First name', 'Last name']
['booker12', '9012', 'Rachel', 'Booker']
['grey07', '2070', 'Laura', 'Grey']
```

We are getting one list for each record. The list contains strings. If a data value is a number, we should perform a type conversion: e.g., `int(line[1])`.

2. [**Suggested**] Using third-party Python packages:
    - several options out there
    - we will see `pandas` later on

TSV file (`.tsv`)

A *Tab-Separated Values* (**TSV**) is just a CSV using the tab character (`\t`) as separator (instead of comma).

Example: `myfile.tsv`

```
Username  Identifier  First name  Last name
booker12  9012  Rachel  Booker
grey07  2070  Laura Grey
```

Remarks:

- Main benefit wrt CSV: the tab character is more convenient than comma or semicolon because it is rarely used with real-world data. Hence, it is more robust, less ambiguous, delimiter.
- The tab character should help visually align and space the data. It fails to do it in most cases when different values have too different lengths.

TSV file (`.csv`): parsing

1. Manually: not hard but a bit raw

In [46]:
```
data = open('myfile.tsv', 'r').read() # you have the prev example file!
for line in data.split('\n'):
    print(line.split('\t'))
```

```
['Username', 'Identifier', 'First name', 'Last name']
['booker12', '9012', 'Rachel', 'Booker']
['grey07', '2070', 'Laura', 'Grey']
```

We are getting one list for each record. The list contains strings. If a data value is a number, we should perform a type conversion: e.g., `int(line[1])`.

2. [**Suggested**] Using third-party Python packages:
    - several options out there
    - we will see `pandas` later on

JSON file format

JavaScript Object Notation (**JSON**):

- is an open standard file format
- often used for data interchange over the network or across different programming languages
- its syntax derives from JavaScript, the programming language supported by browser
- luckily its syntax is quite similar to what your would see when printing Python data
- is like a Python dictionary, recursively containing:
    - scalar types: `int`s, `float`s, `bool`s, `str`s, `None`
    - non-scalar types: list, tuple, dictionary
- to convert other Python types to JSON we will have to battle a bit

JSON file format: example

Example:

```
{
  "users": [
    {
      "Username":"booker12",
      "Identifier":9012,
      "First name":"Rachel",
      "Last name": "Booker"
    },
    {
      "Username":"grey07",
      "Identifier":2070,
      "First name":"Laura",
      "Last name": "Grey"
    }
  ]
}
```

Remarks:

- In our example, we have a dictionary containing a key `users`, whose associated value is a list of users, where each user is a dictionary with the key-value pairs representing the different user attributes.
- Notice that key `Identifier` has a value that is an integer (not a string as the other attributes!). Hence, we can preserve (at least some) data types.
- Differently from TXT and CSV, we can have a quite complex nested structure.
- Most programming languages have the data types supported by JSON, making it a convenient format to pass data over the nework (different machines) or across programs (same machine with programs written in different ways)

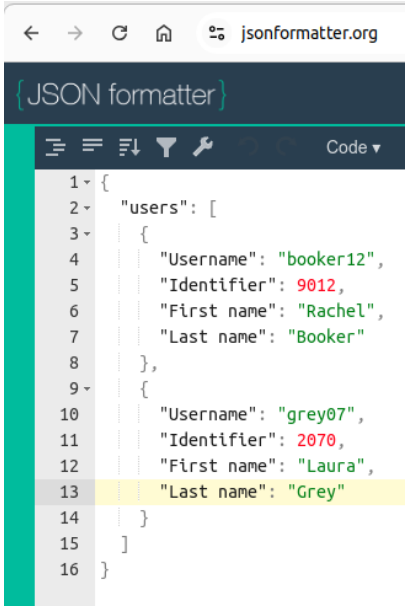
JSON file format: example (cont'd)

Notice that spacing is not important for the format. Indeed, this would be equivalent:

```
{"users": [{"Username":"booker12","Identifier":9012,"First name":"Rachel","Last name": "Booker"},
{"Username":"grey07","Identifier":2070,"First name":"Laura","Last name": "Grey"}]}
```

Nonetheless, it would be less readable for a human. If you get a poorly formatted JSON, you can use some tools to format it properly and make it human readable:

- editor extensions: e.g., "Format" in VSCode
- websites: e.g., https://jsonformatter.org/



JSON file format: parsing

1. Manually: *doable but do not even try... almost impossible!*

2. Python `json` package (pre-installed):

In [49]:
```
# let suppose you get this from a network request
jsondata = '{"users": [{"Username":"booker12","Identifier":9012,"First
import json
data = json.loads(jsondata) # we get a Python dictionary
# let us print the username of the first user
print(data['users'][0]['Username'])
```

```
booker12
```

To navigate the dictionary obtained from a JSON, you need to know its structure. The data provider should give you this information. Otherwise, you can guess it by looking at a first few bits of its textual representation.

3. Most third-party Python packages support JSON. We have already seen that `requests` can automatically parse the JSON data using `response.json()`. Later on, we will introduce `pandas` that supports it as well.

JSON file format: cons

JSON is nowadays standard way of exchange data. Since in the end, it can be stored as a textual file, it is quite convenient for most use cases. For instance, a Jupyter notebook (file `.ipynb`) is a JSON file.

However, it comes with some downsides:

1. If we build it by hand, we need to escape `"` if we use it inside our strings.
2. It is not efficient: it does not have by design compression
3. It is not *secure*: it does not have by design encryption
4. It cannot store raw data: e.g., to store an image into a JSON we have to use a special encoding (e.g., Base64)
5. It cannot be easily inspected when the dataset is large: when having millions of values, we cannot just open it with a text editor and inspect it since it would fill our memory. We must inspect programmatically (which is fine!).
6. A JSON does not necessarily represent tabular data: spreadsheet programs, such as Excel, do not have a clue on how to deal with a JSON.

XSLX file format (`.xlsx`)

XLSX is a file format that:

- was designed by Microsoft for the spreadsheet application Excel
- went through a standardization process and thus should be fully documented. However, it is not.
- is partially supported even by some third-party spreadsheet applications, such as LibreOffice Calc.
- is not human-readable.
- does not scale: it struggles in the presence of millions of values.

AVOID IT IF YOU CAN

OTHERWISE...RUN... RUN AWAY!

XSLX file format (`.xlsx`): parsing

It is a ~~pain-in-the-ass~~ mess to parse it.

Some third-party Python packages provide some basic support to such file format.
One of these is `pandas` that will be introduced later on.