DSS5201 Data Visualization
Week 3

Yuting Huang

NUS DSDS

2024-08-26

Importing data to Python

This week, we will learn how to read data from external sources to Python.

- CSV files
- Excel files
- JSON files
- Data from the web
    - Click and download
    - Web scraping and APIs

An important pre-requisite of loading data into Python is that we are able to **point to the location** at which the data files are stored.

1. Where am I?
2. Where are my data?

The first question addresses the notion of our **current working directory**.

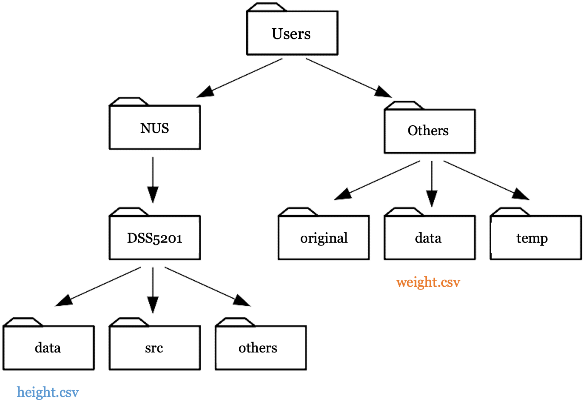
- Typically, the location of the current notebook.
- We can use the `getcwd()` function from the `os` package to return the current location (CWD).

```
import os
current_dir = os.getcwd()
current_dir
```

The second question implies that data are not necessarily stored at the location of the current working directory.

- Absolute path: The exact address of a file on our computer.
- **Relative path**: The address of a file relative to the current working directory.
  - Access files directly in the current path.
  - Use two dots (`..`) to denote "one level up in the directory hierarchy".
  - Use one dot (`.`) to denote "the current directory".

Use relative path in all code you write. This allows you to share your scripts and data files easily with others.



Let's say the current working directory is `/Users/NUS/DSS5201/src`.

- To access the **height** data: `../data/height.csv`.
- To access the **weight** data: `../../../Others/data/weight.csv`.

```
|-- DSS5201
      |-- src
      |-- data
```

We will strictly follow this practice:

- Create a main folder titled **DSS5201**.
    - Within DSS5201, a sub-folder named `src` to store all Python scripts and notebooks.
    - Within DSS5201, a sub-folder named `data` to store all data sets.

Important: The `src` and `data` folders must be position at the same hierarchical level within DSS5201.

Use relative path in all code you write.

Memory requirements for Python objects.

Before we read in the data, remember that Python stores all its objects using physical memory.

- Important to be aware of how much memory is being used in your workspace.
- Especially when reading in or creating a new (large) data set.
- It is often useful to do back-of-the-envelop calculation of how much memory the object will occupy in the workspace.

Suppose we have a data set with 1,500,000 rows and 120 columns, all of which are numeric data.

- Roughly, on modern computers, integers are 4 bytes, numerics are 8 bytes, and character data are usually 1 byte per character.
- Given that, we can do the following calculation:

  $1500000 \times 120 \times 8$ bytes $= 1440000000$ bytes ≈ 1.34 GB
- Most computer these days have at least that much of RAM. But you still need to be aware of
  - Other programs running on your computer, using up RAM at the same time; and
  - Other objects in the current workspace, taking up RAM at the same time.

If you do not have enough RAM, the computer will freeze up.

- Usually an unpleasant experience that requires you to kill the program (the best case scenario), or
- ... reboot your computer.

So make sure you understand the memory requirements before reading in or creating large data sets.

Pre-requisites: Data sets

We'll also need a couple of files from Canvas.

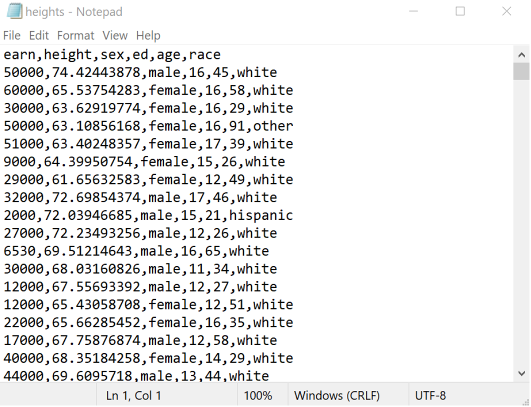
Download and store them in the `data` folder, using our standard folder hierarchy.

- `wk3_csv_01.csv`
- `wk3_height.csv`
- `wk3_excel_01.xlsx`
- `wk3_UNESCAP_population.xlsx`
- `wk3_BusArrival.json`

CSV stands for **comma-separated values**.

- These files are in fact just text files, with
  - an optional header, listing the column names.
  - each observation separated by commas within each row.
- CSV is the easiest format to read into Python.

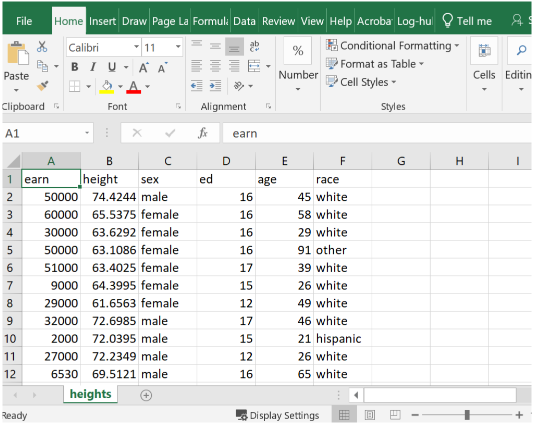
A `.csv` file, opened in a text editor:




```
earn,height,sex,ed,age,race
50000,74.42443878,male,16,45,white
60000,65.53754283,female,16,58,white
30000,63.62919774,female,16,29,white
50000,63.10856168,female,16,91,other
51000,63.40248357,female,17,39,white
9000,64.39950754,female,15,26,white
29000,61.65632583,female,12,49,white
32000,72.69854374,male,17,46,white
2000,72.03946685,male,15,21,hispanic
27000,72.23493256,male,12,26,white
6530,69.51214643,male,16,65,white
30000,68.03160826,male,11,34,white
12000,67.55693392,male,12,27,white
12000,65.43058708,female,12,51,white
22000,65.66285452,female,16,35,white
17000,67.75876874,male,12,58,white
40000,68.35184258,female,14,29,white
44000,69.6095718,male,13,44,white
```

What does a CSV file look like?

Here is the same file opened in Excel:



	A	B	C	D	E	F
1	earn	height	sex	ed	age	race
2	50000	74.4244	male	16	45	white
3	60000	65.5375	female	16	58	white
4	30000	63.6292	female	16	29	white
5	50000	63.1086	female	16	91	other
6	51000	63.4025	female	17	39	white
7	9000	64.3995	female	15	26	white
8	29000	61.6563	female	12	49	white
9	32000	72.6985	male	17	46	white
10	2000	72.0395	male	15	21	hispanic
11	27000	72.2349	male	12	26	white
12	6530	69.5121	male	16	65	white

Read in the CSV file

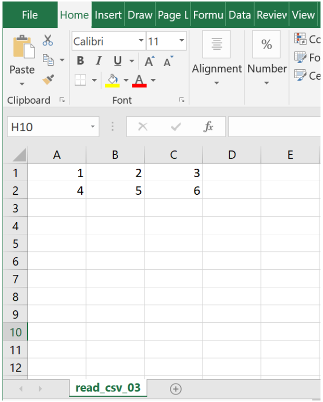
The command to read a CSV file is `pd.read_csv()`. The main arguments are:

- `file`: the file name.
- `header`: the row number containing column labels (zero-indexed).
- `names`: the sequence of column labels to apply.
- `skiprows`: number of lines at the beginning to skip.
- `na_values`: specify the strings to recognize as `NA` or `NaN`.

The full documentation of `pd.read_csv()` can be found here.

Example: A simple CSV file

- Take a first look at the data.
- 2 rows $\times$ 3 columns.
- The data set has no header.



- Let's read a CSV, `wk3_csv_01.csv` into the environment.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
df = pd.read_csv("../data/wk3_csv_01.csv",
                 header = None, names = ["a", "b", "c"])
df
```

	a	b	c
0	1	2	3
1	4	5	6

Example: Education, height, and income

The `height.csv` contains information on 1192 individuals.

- Take a look at the data, you will find that it contains 6 columns and 1 header.
- Hence, we read in the data in the following way:

```
height = pd.read_csv("../data/wk3_height.csv", header = 0)
height.head(3)
```

	earn	height	sex	ed	age	race
0	50000.0	74.424439	male	16	45	white
1	60000.0	65.537543	female	16	58	white
2	30000.0	63.629198	female	16	29	white

The argument `header = 0` indicates that we'd use the first row (index $= 0$) as the header.

1. What type has each column been read in as?

```
height.dtypes
```

```
earn       float64
height     float64
sex         object
ed           int64
age          int64
race        object
dtype: object
```

- `sex` and `race` has been read in as text data (with the `object` data type).
- We can convert them into categorical.

```
height["sex"] = height["sex"].astype("category")
height["race"] = height["race"].astype("category")
height.dtypes
```

```
earn        float64
height      float64
sex        category
ed            int64
age           int64
race       category
dtype: object
```

2. race is a categorical variable. What are the different races that have been read in?
   - `.cat.categories` accesses the levels of a column in the data frame.
   - Alternatively, we can use the `unique()` method.

```
# Method 1
race_levels1 = height["race"].cat.categories
race_levels1
```

```
Index(['black', 'hispanic', 'other', 'white'], dtype='object')
```

```
# Method 2
race_levels2 = height["race"].unique()
race_levels2
```

```
['white', 'other', 'hispanic', 'black']
Categories (4, object): ['black', 'hispanic', 'other', 'white']
```

3. Are there any missing values in the data?

- `.isna()` returns a `DataFrame` of the same shape as the original data frame, with `True` for missing values and `False` for non-missing values.
- `.sum()` sums up the number of `True` per column.

```
missing = height.isna().sum()
missing
```

```
earn      0
height    0
sex       0
ed        0
age       0
race      0
dtype: int64
```

4. We can compute **summary statistics** for earnings:

```
# Summary statistics for the "earn" column
earn_sum = height["earn"].describe()
earn_sum
```

```
count       1192.000000
mean       23154.773490
std        19472.296925
min          200.000000
25%        10000.000000
50%        20000.000000
75%        30000.000000
max       200000.000000
Name: earn, dtype: float64
```

5. We can compute **group statistics** for earnings:

```
# Group statistics: mean "earn" by "sex"
earn_mean_by_sex = height.groupby("sex")["earn"].mean()
earn_mean_by_sex
```

```
sex
female    18280.195051
male      29786.130693
Name: earn, dtype: float64
```

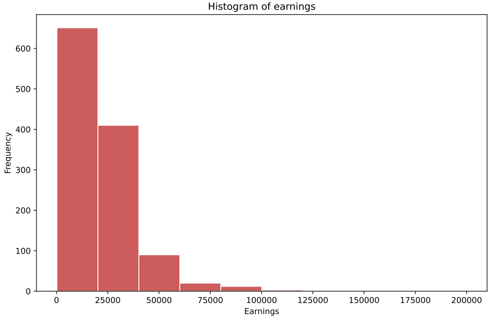
Let's visualize income earned by individuals in this data set.

- `pandas` relies on `matplotlib` as its default plotting backend.
- That's why we loaded the `matplotlib` library at the very beginning.

We will use a **histogram** to visualize **quantitative** variables like `earn`.

- It divides the range of values into bins, then counts the number of values that fall into each bin.

```python
# Create a histogram
height["earn"].plot(kind = "hist",
                    title = "Histogram of earnings", xlabel = "Earnings",
                    color = "indianred", edgecolor = "white")
# Display the plot
plt.show()
```



Histogram (code explained)

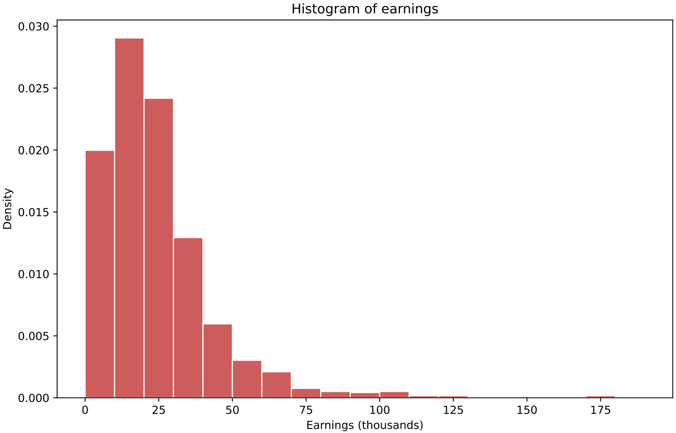
```
# Create a histogram
height["earn"].plot(kind = "hist",
                    title = "Histogram of earnings", xlabel = "Earnings",
                    color = "indianred", edgecolor = "white")
# Display the plot
plt.show()
```

- `kind = "hist"` specifies that we want to create a histogram.
- By default, the height of each bar represents frequencies. We can modify this behavior by setting an additional argument `density = True`.
- `title` and `xlabel` set the title of the histogram and the label of the x-axis, respectively.
- `color` and `edgecolor` set the color of the bars and the bar borders, respectively.
- We need `plt.show()` to display the plot we just created.

1. The bins correspond to intervals of width 20,000. We can modify it to bins of 10,000.
2. Instead of frequencies, we can represent probability density on the vertical axis.

```
# Transformation
height["earn_thousands"] = height["earn"] / 1000
# Create the histogram
height["earn_thousands"].plot(
    kind = "hist", density = True, bins = range(0, 200, 10),
    title = "Histogram of earnings",
    xlabel = "Earnings (thousands)", ylabel = "Density",
    color = "indianred", edgecolor = "white")
# Show the plot
plt.show()
```

Histogram


Histogram of earnings
Density
0.030
0.025
0.020
0.015
0.010
0.005
0.000
0
25
50
75
100
125
150
175
Earnings (thousands)

Who are the high-earning individuals – earn more than 100,000 per year?

- We can use boolean conditions to filter those individual rows.

```
# Filter for individuals earning more than 100,000
high_earners = height[height["earn_thousands"] > 100]
high_earners.head()
```

	earn	height	sex	ed	age	race	earn_thousands
174	125000.0	74.340622	male	18	45	white	125.0
202	170000.0	71.010034	male	18	45	white	170.0
339	175000.0	70.589553	male	16	48	white	175.0
356	148000.0	66.740195	male	18	38	white	148.0
376	110000.0	65.965038	male	18	37	white	110.0

Recap

- Remember that you should inspect your data before and after you read them in.
- Try to think of as many ways in which it could have gone wrong and check.
- As we covered here, you should at least consider the following:
    - Correct number of rows and columns.
    - Column variables read in with the correct class type.
    - Missing values.

In Python, we use the pandas library along with openpyxl (for .xlsx) or xlrd (for .xls) to read Excel files.

- Let's read in our first Excel files into the workspace.

```
import openpyxl
df_xlsx = pd.read_excel("../data/wk3_excel_01.xlsx")
df_xlsx.head(5)
```

```
   Table 1  Unnamed: 1 Unnamed: 2  Unnamed: 3 Unnamed: 4
0      NaN         NaN        NaN         NaN        NaN
1      NaN         NaN        NaN         NaN        NaN
2      NaN         NaN        NaN         NaN        NaN
3      NaN         NaN        NaN         NaN        NaN
4      NaN         NaN          a         1.0          m
```

What's wrong with the data we read in?

- Open up the file in Excel. The data seem to be "floating" in the center of the worksheet.
- In this case, the `read_excel()` function needs a little help.
  - Use `skiprows` and `usecols` to tell Python the location of the data.
  - The data have no header. We can specify the column names via the `names` argument.

```
df_xlsx = pd.read_excel(
    "../data/wk3_excel_01.xlsx",
    skiprows = 5, usecols = "C:E", names = ["col1", "col2", "col3"])
df_xlsx.head(5)
```

```
  col1  col2 col3
0    b     2    m
1    c     3    m
```

The excel file `wk3_UNESCAP_population.xlsx` contains the population for selected Asia-Pacific countries from 2010 to 2015.

- The counts are broken down by age group and gender.
- Data for each age group and gender are stored in different spreadsheets.

Suppose we want to read in data for **female aged 0-4years**.

- That is, the third spreadsheet named `Pop, female, 0-4 years`.

- Reading in the third spreadsheet:

```
female_0_4 = pd.read_excel(
    "../data/wk3_UNESCAP_population.xlsx", sheet_name = 3)
female_0_4.head()
```

	e_fname	Y2010	Y2011	Y2012	Y2013	Y2014	Y2015
0	Afghanistan	2189	2238	2287	2334	2371	2393
1	Armenia	81	80	81	84	88	91
2	Australia	661	671	686	704	723	740
3	Azerbaijan	292	288	286	286	287	290
4	Bangladesh	8081	8033	7961	7868	7766	7672

JSON (JavaScript Object Notation) is a standard **text-based format** for storing structured data.

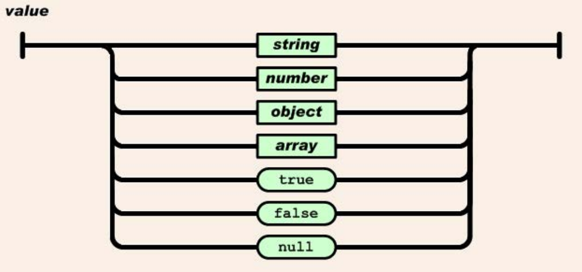
- A very popular format for data interchange on the internet.
- The full description of the format can be found at http://www.json.org/.
- The syntax is easy for humans to read and write, and for computers to parse and generate.

JSON description

- JSON is built on two structures:
  - An **object** is an unordered collection of name/value pairs.
  - An **array** is an ordered list of values.
- By repeatedly stacking these structures on top of one another, we will be able to store quite complex data structures.

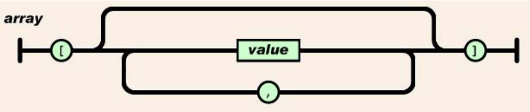
```
object
        {}
        { members }
members
        pair
        pair , members
pair
        string : value
array
        []
        [ elements ]
elements
        value
        value , elements
value
        string
        number
        object
        array
        true
        false
        null
```

A **value** can be a string (in double quotes), a number, an object, an array, or a true or false or null.



An **array** is an ordered collection of values, represented as Python lists.

- Surrounded with square brackets, starts with `[` and ends with `]`
- Values are separated by a comma `,`

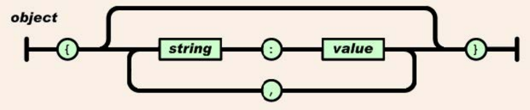


Example:

- `[12, 3, 7]` is an JSON array with three elements, all are numbers.
- `["Hello", 3, 7]` is an array with mixed types.

An **object** is an unordered set of name/value pairs, represented as dictionaries.

- Surrounded with curly braces, starts with { and ends with }
- Each name is followed by a colon : and the name/value pairs are separated by a comma ,



Example:

- `{"fruit": "Apple"}` is a valid JSON object with a single name-value pair.
- `{"fruit": "Apple", "price": 2.03}` is also valid.
  - Two name-value pairs. The names are "fruit" and "price".

Let's read in real-time data on minute-by-minute bus arrival times from every bus stop in Singapore.

- The data were obtained from the LTA Data Mall and available as `wk3_BusArrival.json` on Canvas.

```
import json
file_path = "../data/wk3_BusArrival.json"
# Read the JSON file
with open(file_path, "r") as file:
    bus_arrival = json.load(file)
# Type of the parsed data
type(bus_arrival)
```

```
<class 'dict'>
```

Example: Bus arrival times in Singapore

```
bus_arrival
```


```
{'odata.metadata': 'http://datamall2.mytransport.sg/ltaodataservice/$metadata#BusArrivalv2/@Element',
 'BusStopCode': '20251',
 'Services': [{'ServiceNo': '176',
   'Operator': 'SMRT',
   'NextBus': {'Origin': '10009',
    'Destination': '45009',
    'EstArrival': '2020-02-12T14:09:11+08:00',
    'Lat': '1.301219',
    'Long': '103.762202'}},
  {'ServiceNo': '78',
   'Operator': 'TTS',
   'NextBus': {'Origin': '28009',
    'Destination': '28009',
    'EstArrival': '2020-02-12T14:09:09+08:00',
    'Lat': '1.30693',
    'Long': '103.73333'}}]}
```

- Extract information and convert it into a useful data frame.

```
df_bus_arrival = pd.json_normalize(data = bus_arrival,
                                   record_path = ["Services"])
df_bus_arrival.head()
```

```
  ServiceNo Operator  ... NextBus.Lat NextBus.Long
0       176     SMRT  ...    1.301219   103.762202
1        78      TTS  ...     1.30693    103.73333

[2 rows x 7 columns]
```

Data from the web

In the simplest case, the data you need are ready on the internet in a tabular format.

- Just click and download.

If data are **not** downloadable in a tabular format:

- Server side (back-end): Web scrapping.
- Client side (front-end): Using APIs.

#MakeoverMonday

Improving How We Visualize and Analyze Data,
One Chart at a Time

Makeover Monday is a weekly social data project for the online learning community.

- Raw data set(s) and a related article every week.
- Emphasize on the understanding of how to summarize and arrange data to make meaningful visualizations.
- Full list of data sets can be found on https://makeovermonday.co.uk/

Let's explore the data set posted on Aug 14, 2023.

- A data set on energy use per person across countries from 1998 to 2022.
- You can find an overview of the data at: *https://data.world/makeovermonday/2023w33*
- **Download** the following data set and put it in your `data` folder:

  `per-capita-energy-use.csv`

```python
# Read in the data
energy = pd.read_csv("../data/per-capita-energy-use.csv", header = 0,
                     names = ["Entity", "Code", "Year", "Energy_use"])
energy.head(3)
```

	Entity	Code	Year	Energy_use
0	Afghanistan	AFG	1980	623.92865
1	Afghanistan	AFG	1981	786.83690
2	Afghanistan	AFG	1982	926.65125

```python
# Shape of the data set
energy.shape
```

```
(10602, 4)
```

- From the output, we know that there are missing values.

```
# Number of missing values per column
energy.isna().sum()
```

```
Entity          0
Code          622
Year            0
Energy_use      0
dtype: int64
```

- Let's take a further look at the missing values.
- These are observations associated with continents or groups of countries.

```
# Filter and examine rows with missing values
missing_data = energy[energy.isna().any(axis = 1)]
missing_data.head(3)
```

```
    Entity Code  Year  Energy_use
42  Africa  NaN  1965   2228.4226
43  Africa  NaN  1966   2275.4866
44  Africa  NaN  1967   2241.3489
```

- We can remove rows with missing values in these columns with `dropna()`.
- End up with a data frame with 9980 rows and 4 columns.

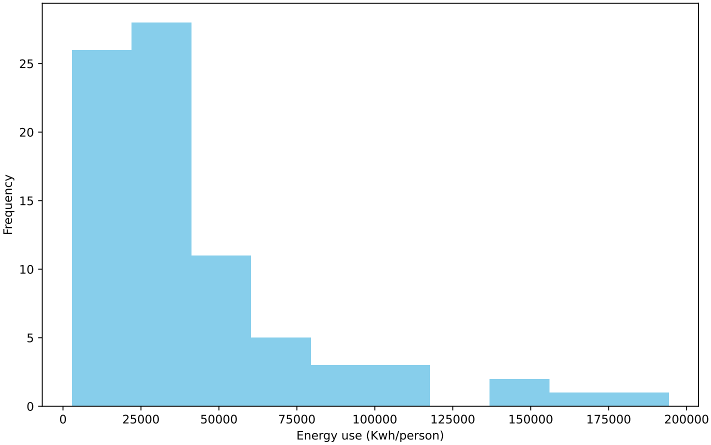
```
energy.dropna(subset = ["Code"], inplace = True)
energy.shape
```

```
(9980, 4)
```

- A **histogram** for per-capita energy use in 2022.

```
# Filter rows for 2022
energy22 = energy[energy["Year"] == 2022]
# Create a histogram
energy22["Energy_use"].plot(kind = "hist", color = "skyblue",
                            xlabel = "Energy use (Kwh/person)")
# Show the plot
plt.show()
```

Histogram



Frequency
0
5
10
15
20
25
0
25000
50000
75000
100000
125000
150000
175000
200000
Energy use (Kwh/person)

Often times, data are present on a website, but are not downloadable in a tabular format.

- It's possible to grab that information too.

There are two ways that web contents gets rendered in our browser.

1. Server side (back-end):

- The scripts that build the website are on a host server that processes information, and embed it in the website's HTML.
- **Challenges:** Find the correct CSS selectors (e.g., table); iterate through dynamic pages (e.g., "Next Page" or "Show More" tabs).
- **Key concepts:** CSS, HTML.

2. Client side (front-end):

- The website contains an empty template of HTML and CSS.
- When we visit the page, the browser send request to the host server.
- Then the server sends back a *response* script, with the information we want.
- **Challenges**: Find the API endpoints and send the correct request.
- **Key concepts**: APIs, API endpoints.

Web scraping is one of many ways to get data.

- Process of converting semi-structure data from the internet into a structured (tabular) data set.
- Useful when information is already online, but not available in a nice format.

It includes three steps.

1. Get the URL(s) to scrape.
2. Download information from the link and store them in a systematic way.
3. Parse data from the downloading content.

We can use the `read_html()` method in `pandas` to read simple tables from the web.

- Automatically parses all tables in an HTML document and converts them into data frames.
- Limited flexibility and less control over the parsing process.

For more complex HTML pages, we can use packages such as `beautiful soup`.

- The official documentation for `beautiful soup` can be found here.
- A lot more flexible.
- Requires understanding about HTML and CSS for more complex scraping tasks.

Let's see an example:

- Wikipedia table of the least-polluted cities by PM2.5.

  https://en.wikipedia.org/wiki/List_of_least-polluted_cities_by_particulate_matter_concentration.

```
url = "https://en.wikipedia.org/wiki/List_of_least-polluted_cities_by_part
tables = pd.read_html(url) # Returns a list of data frame(s)
df = tables[0]
df.head(3)
```

```
   Rank Country / Region           City  Average PM2.5 (ug/m3)
0     1       Switzerland        Zürich                   0.49
1     2         Australia         Perth                   1.61
2     3      South Africa  Richards Bay                   2.38
```

Visit the Wikipedia page on the 2024 Summer Olympics, which concluded earlier this month.

https://en.wikipedia.org/wiki/2024_Summer_Olympics

1. How many tables are there on the page?
2. Extract tables on
   - Host city election, and
   - Number of atheletes by National Olympic Committees (NOCs).

Legalities depend a lot on where you live.

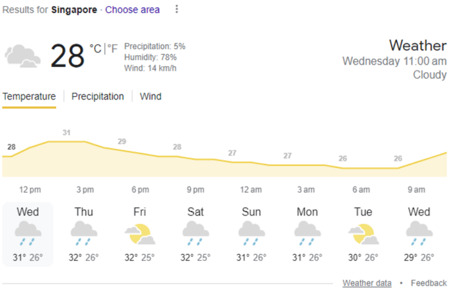
- If data are public, non-personal, and factual, you are likely okay.
- If data are not public, non-personal, or factual, or you are scraping the data specifically to make money with it, you will need to talk to a lawyer.
- `robots.txt` on the page.

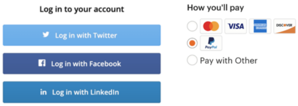
Be respectful of the resources of the server hosting the pages that you are scraping.

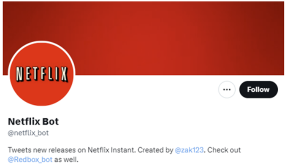
- Seek permission before scraping.
- Take it slowly (set waiting times between requests).
- Collect only once (download once, not every time you run the script).

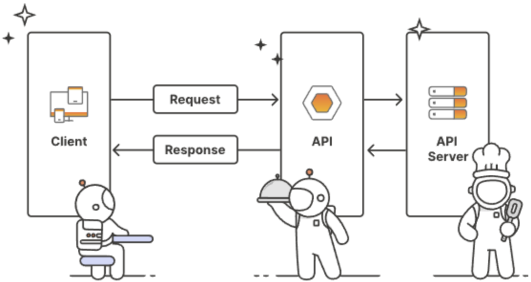
Next, let's turn to scraping data that are rendered on the **client side**.

- Using APIs.
- When available, this approach is typically easier than scraping data directly from the web.









Websites or Apps that are built using **client-side framework** typically involve:

- Visit the URL that contains a template of static content.
- The browser sends a **request** to the host server.
- If the request is valid, the server issues a **response** that fetches the necessary data and renders the page dynamically on the browser.

All of these take place through the host application's **API (Application Program Interface)**.

An **API** is a set of rules that allow different pieces of software to communicate with each other.

- **Server:** A powerful computer that runs an API.
- **Client:** A program that exchanges data with a server through an API.
- **Protocol:** The etiquette underlying how computers talk to each other (e.g., HTTP).
- **Methods:** The "verbs" that clients use to talk with a server.
- **Request:** What the client asks of the server.
- **Response:** The server's response. This includes:
  - A status code (e.g., 404 = if not found; 200 if successful).
  - Header (meta-information about the response).
  - Body (actual content in the response).

In the case of web APIs, we can access information directly from the API database if we can specify the correct URL.

- These are known as **API endpoints**.
- Similar to normal website URLs, except that it is much less visually appealing.
- For example, Singapore's real-time PM2.5 readings across regions: *https://api-open.data.gov.sg/v2/real-time/api/pm25*


```
{"code":0,"data":{"regionMetadata":[{"name":"west","labelLocation":
{"latitude":1.35735,"longitude":103.7}},{"name":"east","labelLocation":
{"latitude":1.35735,"longitude":103.94}},{"name":"central","labelLocation":
{"latitude":1.35735,"longitude":103.82}},{"name":"south","labelLocation":
{"latitude":1.29587,"longitude":103.82}},{"name":"north","labelLocation":
{"latitude":1.41803,"longitude":103.82}}],"items":[{"date":"2024-08-21","updatedTimestamp":"2024-08-
21T15:30:34+08:00","timestamp":"2024-08-21T15:00:00+08:00","readings":{"pm25_one_hourly":
{"west":23,"east":21,"central":12,"south":20,"north":18}}}]},"errorMsg":""}
```

APIs has become an increasingly popular method for collecting data.

There are typically two ways to collect data with API in Python.

- Use an API wrapper that comes with functions that call data from the API.
- If wrappers are not available, we need to set up the query ourselves.
    - Requires a bit more work since we will need to find the correct **API endpoints** and look up the documentation for the right **query parameters**.

Some web services provide functions which send our query to the server and format the response.

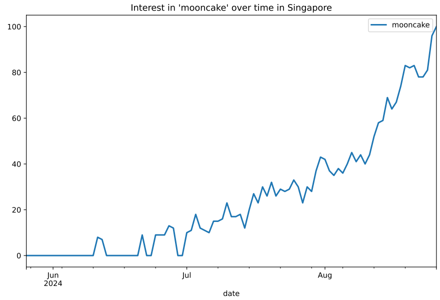
- Examples: `pytrends` for Google Trends.

```
from pytrends.request import TrendReq
pytrends = TrendReq(hl = "en-US", tz = 360)
kw_list = ["mooncake"]
pytrends.build_payload(kw_list, timeframe = "today 3-m", geo = "SG")
results = pytrends.interest_over_time()
results.head(2)
```

```
            mooncake  isPartial
date
2024-05-26         0      False
2024-05-27         0      False
```

- A **line chart** for time-series data.

```
results.drop(columns = ["isPartial"], inplace = True)
results.plot(title = "Interest in 'mooncake' over time in Singapore",
             linewidth = 2)
plt.show()
```



Now we are going to create the API request ourselves.

- This is important because often times, there is no package that calls to the API of interest.
- We have to set up the query manually.

Querying an API

In the following, we will query PM2.5 readings for major regions in Singapore.
https://beta.data.gov.sg/datasets/d_e1058d6974c877257e32048ab128ad83/view

- The data update hourly from NEA.
- Read the **API documentation** section and look for the request URL.

API documentation

GET /pm25 PM 2.5

https://api-open.data.gov.sg/v2/real-time/api/pm25

- Updated hourly from NEA.
- Readings are provided for each major region in Singapore

- Filter for specific date or date-time by providing `date` in query parameter.
  - use YYYY-MM-DD format to retrieve all of the readings for that day
  - use YYYY-MM-DDTHH:mm:ss to retrieve the latest readings at that moment in time
  - example: `?date=2024-07-16` or `?date=2024-07-16T23:59:00`

- The `region_metadata` field in the response provides longitude/latitude information for the regions. You can use that to place the readings on a map.

```python
import requests
url = "https://api-open.data.gov.sg/v2/real-time/api/pm25"
query_params = {"date": "2024-08-03"}
response = requests.get(url, query_params)
results = response.json()
df = pd.json_normalize(results["data"]["items"])
df.head()
```

```
         date  ... readings.pm25_one_hourly.north
0  2024-08-03  ...                              5
1  2024-08-03  ...                              7
2  2024-08-03  ...                             11
3  2024-08-03  ...                             14
4  2024-08-03  ...                             13

[5 rows x 8 columns]
```

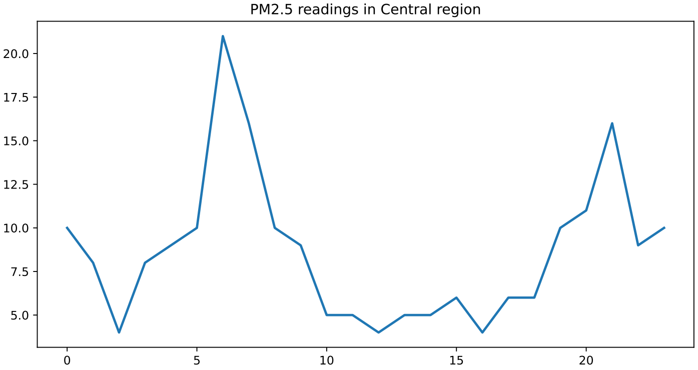
```
# Column names
df.columns
```

```
Index(['date', 'updatedTimestamp', 'timestamp',
       'readings.pm25_one_hourly.west', 'readings.pm25_one_hourly.east',
       'readings.pm25_one_hourly.central', 'readings.pm25_one_hourly.south
       'readings.pm25_one_hourly.north'],
      dtype='object')
```

```
# Define column names
col_names = ["date", "updated_timestamp", "timestamp",
             "west", "east", "central", "south", "north"]
df.columns = col_names
```

- Again, we use a **line chart** to visualize the PM2.5 reading in the Central region.
- This time we will `matplotlib` syntax.

```
# Extract hours from timestamp
from datetime import datetime
df["hours"] = pd.to_datetime(df["updated_timestamp"]).dt.hour
# Set figure size as needed
plt.figure(figsize = (10, 5))
# Plot
plt.plot(df["hours"], df["central"], linewidth = 2)
plt.title("PM2.5 readings in the central region")
# Show the plot
plt.show()
```

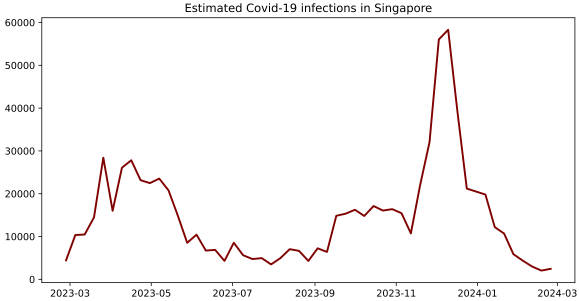

PM2.5 readings in Central region
20.0
17.5
15.0
12.5
10.0
7.5
5.0
0
5
10
15
20

Your turn: Covid-19 infections

There are 4000+ data sets available on data.gov.sg.

In the following exercise, we will practice querying data about Covid-19 infections in Singapore by epi-week.

1. Read the API documentation on the page and create your own queries.
2. Visualize the data in a line chart.



Pandas vs. matplotlib

Both `pandas` and `matplotlib` are powerful tools for data visualization in Python.

- Serve different purposes and offer different levels of customization and flexibility.
- `Pandas`: Quick and easy plotting directly from DataFrames and Series.
- `Matplotlib`: More flexible and highly customizable: Customization options for almost every aspect of a plot!
  - Also, it offers more plot types.
  - Relatively steeper learning curve.

Other visualization libraries

In addition to `pandas` and `matplotlib`, there are several popular and powerful visualization libraries in Python.

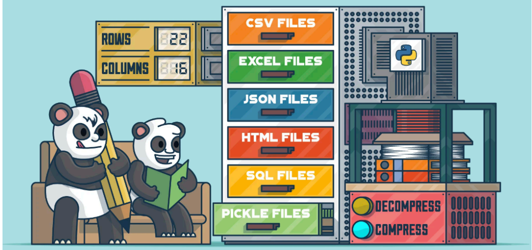
- `Seaborn`: Built on top of `matplotlib`.
- `plotnine`: Implementation of the Grammar of Graphics in Python.
- `plotly` and `lets-plot`: Interactive graphing library for interactive plots and dashboards.

There are many useful visualization libraries. We shall cover some of them in the future.

We learn about importing data from different formats and sources:

1. CSV file.
2. Excel file.
3. JSON objects from text files and JSON files.
4. Data from the web.
    - Click and download.
    - Web scraping and APIs.

Also a few more ways to manipulate and visualize data.



- Importing data becomes complicated when data is not stored in a friendly format.
- When reading data from the web, we need to have some creativity to identify patterns or keywords.
- The patterns are unlikely to be the same every time, but the experience you gather will help you along.