Data Sourcing

The present lecture is about Pandas' **I/O** (https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html) API (Input/Output)

We are going to cover loading data:

- from a CSV (`read_csv` (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html))
- from an API (`requests` (https://pypi.org/project/requests/))
- with SQL queries (`pandas.read_sql` (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_sql.html))
- with Google Big Query (`pandas.read_bgq` (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_gbq.html))
- with Web Scraping (`BeautifulSoup` (https://www.crummy.com/software/BeautifulSoup/bs4/doc/))

We'll talk about building out and enriching an original DataFrame with data from all sources, and discussing which methods are most useful for obtaining data.

🛠️ Let's use Jupyter lab (https://jupyterlab.readthedocs.io/en/stable/)

```
jupyter lab
```

Let's create a new Notebook and start with:

In [ ]:
```
import matplotlib
%matplotlib inline
import numpy as np
import pandas as pd
```

1️⃣ CSV

Let's load the Top Spotify Tracks of 2017 (https://www.kaggle.com/nadintamer/top-tracks-of-2017) dataset into a DataFrame

In [ ]:
```
tracks_df = pd.read_csv('data/spotify_2017.csv')
```

In [ ]:
```
tracks_df.head(2)
```

Out[ ]:

	id	name	artists	danceability	energy	key	loudness	mode
0	7qiZfU4dY1lWllzX7mPBI	Shape of You	Ed Sheeran	0.825	0.652	1.0	-3.183	0.0
1	5CtI0qwDJkDQGwXD1H1cL	Despacito - Remix	Luis Fonsi	0.694	0.815	2.0	-4.328	1.0

2 rows × 21 columns

2️⃣ API

Let's try this Lyrics API (https://lyrics.lewagon.ai/) to enrich our DataFrame with song lyrics for each row.

Make a request in the browser:

```
https://lyrics.lewagon.ai/search?artist=The%20Beatles&title=Come%20together
```

In [ ]:
```
import requests

def fetch_lyrics(artists, title):
    """
    Get lyrics from Seeds Lyrics API. Returns empty string if song not found
    """
    url = f'https://lyrics.lewagon.ai/search?artist={artists}&title={title}'
    response = requests.get(url)
    if response.status_code != 200:
        return ''
    data = response.json()
    return data['lyrics']
```

In [ ]:
```
fetch_lyrics('The Beatles', 'Come Together')[0:100]
```

Out[ ]:
```
'Here come old flat top\nHe come grooving up slowly\nHe got joo joo eyeball\nHe one holy roller\nHe got h'
```

In [ ]:
```
fetch_lyrics('The Beatles', "Wouldn't it be nice")
```

Out[ ]:
```
''
```

💪 Let's do some refactoring and **extract the Python code** from the Notebook.

```
# music.py

# [...] imports

def fetch_lyrics(artists, title):
    # [...] the body from previous slide
```

Then in the notebook you can replace the cell with the function `def`inition with:

```
from music import<SPACE><TAB>
```

In [ ]:
```
from music import fetch_lyrics
```

🤔 What if you **change** the code in `music.py` ?

For instance, make the `fetch_lyrics` return `"NO LYRICS"` if no lyrics are found with the API.

Then run the `fetch_lyrics("The Beatles", "Come not together")` code again in the notebook.

😱 The **old** code is executed! Notebook ignores the changes of `music.py` on the hard drive.

💡 Let's introduce the IPython extension `autoreload` (https://ipython.readthedocs.io/en/stable/config/extensions/autoreload.html)

At the top of the notebook, add this cell. Then **restart** the Kernel and run cells again.

In [ ]:
```
%load_ext autoreload
%autoreload 2
```

Open the `music.py` and **re-save** the file to trigger the `autoreload` of the module.

Then execute the `fetch_lyrics(...)` cell once again. It should pick up the new code!

🙈 OK, back to our Dataframe `tracks_df` .

Let's use this `fetch_lyrics` function to loop over each row of the dataframe and create a new column:

💡 Let's use `pandas.DataFrame.iterrows` (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.iterrows.html) and `pandas.DataFrame.loc` (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.loc.html)

In [ ]:
```
tracks_df['lyrics'] = ""
```

In [ ]:
```
%%time
for index, row in tracks_df.head(3).iterrows():
    print(f"Fetching lyrics for {row['artists']} - {row['name']}")
    lyrics = fetch_lyrics(row['artists'], row['name'])
    tracks_df.loc[index, 'lyrics'] = lyrics
```

In [ ]:
```
print(tracks_df.loc[0, 'lyrics'][0:150])
```

We're good to proceed with our whole DataFrame!

In [ ]:
```
for index, row in tracks_df.iterrows():
    print(f"Fetching lyrics for {row['artists']} - {row['name']}")
    lyrics = fetch_lyrics(row['artists'], row['name'])
    tracks_df.loc[index, 'lyrics'] = lyrics
```

3️⃣ SQL

Next, we'll enrich our data with information from this (https://wagon-public-datasets.s3.amazonaws.com/02-Data-Toolkit/02-Data-Sourcing/music.sqlite) (quite large) SQL file. We're interested in finding out how many `last_fm_listeners` each artist has.

Let's take a moment to explore it in DBeaver and with `sqlite3` !

In [ ]:
```
import pandas as pd
import sqlite3
conn = sqlite3.connect("data/music.sqlite")
```

In [ ]:
```
cursor = conn.cursor()
cursor.execute("SELECT name FROM sqlite_master WHERE type='table'")
print(cursor.fetchall())
```

```
[('artist_info',), ('popularity',)]
```

In [ ]:
```
cursor = conn.cursor()
cursor.execute("SELECT COUNT(*) FROM artist_info")
print(cursor.fetchall())
```

```
[(1466083,)]
```

If we run the following query we see we have **two** entries for Ed Sheeran and that it takes a long time to run 🙄

```
SELECT * FROM artist_info ai
JOIN popularity p on p.mbid = ai.mbid
WHERE artist_mb = "Ed Sheeran"
```

In [ ]:
```
cursor = conn.cursor()
cursor.execute("""
    SELECT artist_mb FROM artist_info ai
    JOIN popularity p on p.mbid = ai.mbid
    WHERE artist_mb = 'Ed Sheeran'
""")
print(len(cursor.fetchall()))
```

```
2
```

We could loop through our artists one by one, but a quicker and simpler solution might be to get all the confirmed large artists (i.e. those with 500k+ million listeners) into a DataFrame and then do our merge!

In [ ]:
```
listens_df = pd.read_sql("""
SELECT artist_mb, listeners_lastfm
FROM artist_info a
JOIN popularity p on p.mbid = a.mbid
WHERE listeners_lastfm > 500000
""", conn)
```

In [ ]:
```
listens_df.head(3)
```

Out[ ]:

	artist_mb	listeners_lastfm
0	Coldplay	5381567.0
1	Radiohead	4732528.0
2	Red Hot Chili Peppers	4620835.0

Now we can merge only the columns we want into our larger DataFrame using our common key 💡

In [ ]:
```
tracks_df = tracks_df.merge(
    listens_df,
    left_on = "artists",
    right_on= "artist_mb",
    how = "left"
)
```

4️⃣ Google BigQuery

As an example, we are going to use the BigQuery `listen_brainz` (https://console.cloud.google.com/marketplace/details/metabrainz/listenbrainz?project=vector-ai-botterill) Dataset to run a query. Here, we get a unique row for each time a user reported listening to a particular song!

This is just one of many (https://console.cloud.google.com/marketplace/browse?filter=solution-type:dataset) available for free on BigQuery!

```
project_id = 'your-project-id-here' # TODO: replace with your own!

sql = """
SELECT artist_name FROM `listenbrainz.listenbrainz.listen`
WHERE listened_at BETWEEN "2017-01-01" AND "2018-01-01"
LIMIT 10
"""
music_brainz_df = pandas_gbq.read_gbq(sql, project_id=project_id)
```

Original package has been **extracted** from `pandas` and needs a separate install.

👉 Documentation (https://pandas-gbq.readthedocs.io/en/latest/)

In [ ]:
```
!pip install --quiet pandas-gbq
```

In [ ]:
```
import pandas_gbq
```

Create (or select) a project in the Google Cloud Console (https://console.cloud.google.com/bigquery). You need a `project_id` .

Again, we'll get out our largest artists from the period we're interested in and then merge it into our `tracks_df`

In [ ]:
```
project_id = 'your-project-id-here' # TODO: replace with your own!

sql = """
SELECT artist_name, COUNT(artist_name) FROM `listenbrainz.listenbrainz.listen`
WHERE listened_at BETWEEN "2017-01-01" AND "2018-01-01"
GROUP BY artist_name
HAVING COUNT(artist_name) > 1000
ORDER BY COUNT(artist_name) DESC
"""
musicbrainz_df = pandas_gbq.read_gbq(sql, project_id=project_id)
```

```
Downloading: 100%|██████████| 2708/2708 [00:00<00:00, 8295.66rows/s]
```

Finally, we can pull this all into our DataFrame, by - again - merging on our common key.

In [ ]:
```
musicbrainz_df.columns = ["artists", "music_brainz_plays"]
```

In [ ]:
```
tracks_df = tracks_df.merge(
    musicbrainz_df,
    on = "artists",
    how = "left"
)
```

For a private BigQuery table, you will need credentials (https://pandas-gbq.readthedocs.io/en/stable/howto/authentication.html) setup

5 Scraping

In the Data Sourcing with Python lecture (https://kitt.lewagon.com/karr/data-lectures.kitt/01-Python_02-Data-Sourcing.slides.html?title=Data+Sourcing&program_id=10#/), we came up with that code to scrape The 50 Best Movies Ever Made (https://www.imdb.com/list/ls055386972/) list. Now, we're going to try scraping Wikipedia to get one final piece of information about our artists - their birthdays.

In [ ]:
```
import requests
from bs4 import BeautifulSoup
import re
```

Whenever scraping, get it working once and then do it for all artists. So, let's take a look at Ed Sheeran's Wikipedia page (https://en.wikipedia.org/wiki/Ed_Sheeran).

In [ ]:
```
# What does our URL look like?
url = "https://en.wikipedia.org/wiki/Ed_Sheeran"
# Get the response
response = requests.get(url)
# Turn it into Soup
soup = BeautifulSoup(response.text, "html.parser")
# Find the right tag
life_info = soup.find("span", style= "display:none")
# Clean up our birthday
clean_birthday = life_info.text.strip()[1:-1]
print(clean_birthday)
```

```
1991-02-17
```

Now let's chain this together into a function:

In [ ]:
```
def birthday_scraper(artist):
    formatted_artist = artist.replace(" ", "_")
    # Wikipedia URL for the artist's page
    url = f"https://en.wikipedia.org/wiki/{formatted_artist}"
    # Send a GET request to fetch the webpage
    try:
        # Get the response
        response = requests.get(url)
        # Turn it into Soup
        soup = BeautifulSoup(response.text, "html.parser")
        # Find the right tag
        life_info = soup.find("span", class_="bday")
        # Clean up our birthday
        clean_birthday = life_info.text
        return clean_birthday
    except:
        return "Inconclusive"
```

A word on the `try ... except ...` construction:

- This is the Pythonic way to catch errors, you will see it regularly in production code.
- Here we use it to handle not finding the artist or their birthday.
- **Don't add a `try ... except ...` construction while you're still developing!**
- It catches all errors ... so **you will never see any error message**, and you won't know what is wrong with your code.
- Only add `try ... except ...` when you already know that your code works.

We can do things with a for loop:

In [ ]:
```
artists_list = list(set(tracks_df["artists"].tolist()))
len(artists_list)
```

Out[ ]: 78

In [ ]:
```
birthdays = []
for artist in artists_list:
    birthday = birthday_scraper(artist)
    birthdays.append(birthday)
```

Then we can put our lists into a dictionary to a DataFrame and merge.

In [ ]:
```
birthdays_df = pd.DataFrame({"artists": artists_list,
                             "birthday": birthdays})

# We can then merge our DataFrame as we did before on "artists"
```

Or we can `map` our new function to a column in our existing DataFrame (though this would be less efficient since we're performing the same operation on artists that appear multiple times 😤)

In [ ]:
```
tracks_df["birthday"] = tracks_df["artists"].map(
    lambda x: birthday_scraper(x)
)
```

What do we notice?

Our scraping has missed quite a few birthdays 😢

We can look at the url formatting to quickly see why - look at Halsey's Wikipedia URL (https://en.wikipedia.org/wiki/Halsey). Even then, The Chainsmokers (https://en.wikipedia.org/wiki/The_Chainsmokers) has no birthday at all!

⚠️ Scraping can be unreliable and time-consuming so is often a last resort when you know there are no SQL databases, CSVs or APIs out there that have your information.

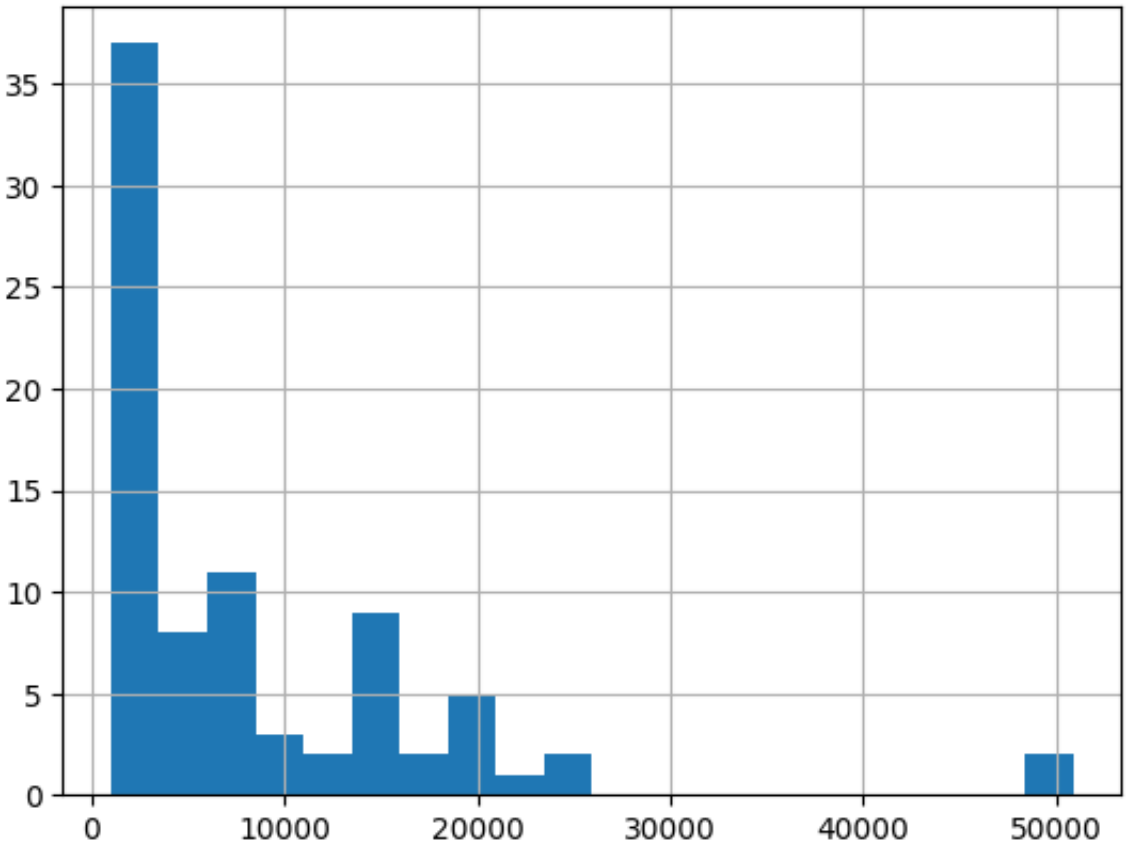
😇 For future reference - if you ever do want information from Wikipedia there is a lovely API for Python (https://wikipedia-api.readthedocs.io/en/latest/README.html) we could have used

6️⃣ Quick plots with pandas

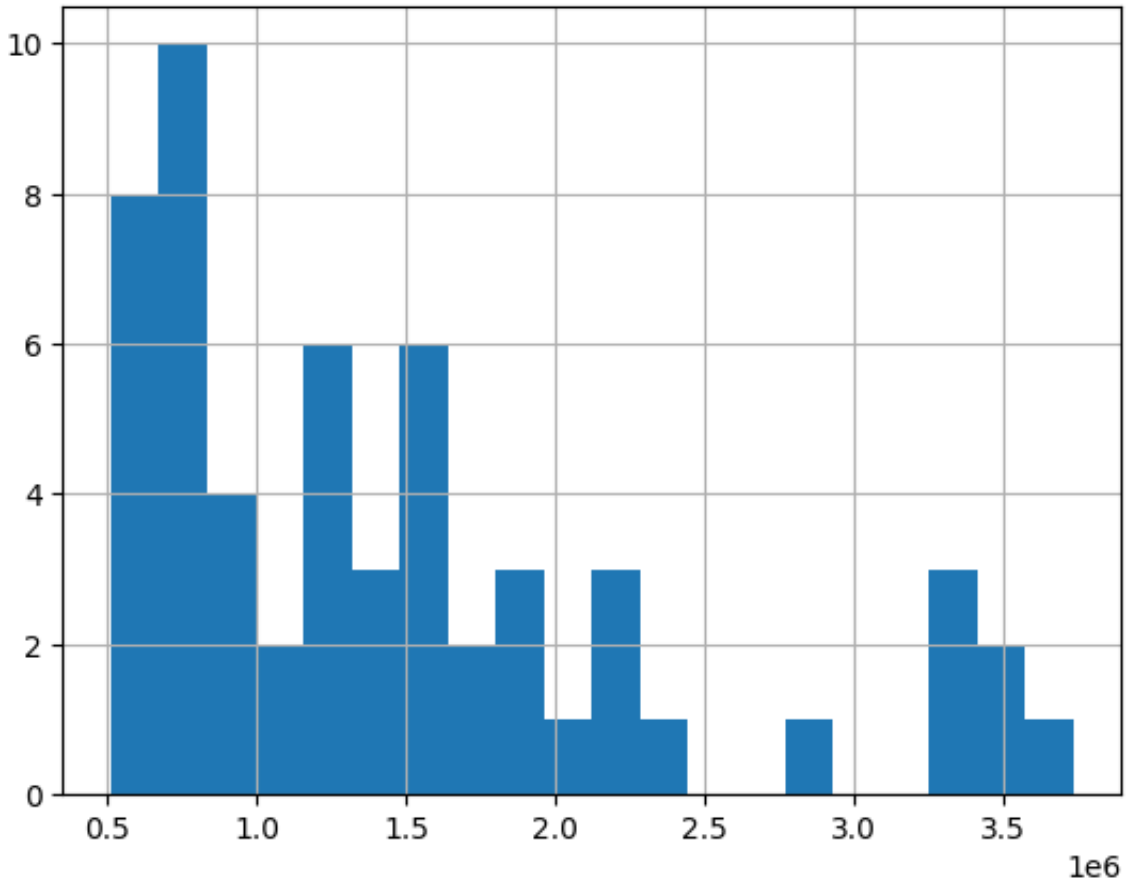
Let's use `pandas.DataFrame.hist` (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.hist.html) to visualize some of the stats we've pulled together!

```
tracks_df["column"].hist()
```

In [ ]:
```
tracks_df["music_brainz_plays"].hist(bins = 20);
```



In [ ]:
```
tracks_df["listeners_lastfm"].hist(bins = 20);
```



We can even do some analysis on birthdays vs plays! Let's get out our valid birthdays using regex (see this really useful regex builder (https://regex-generator.olafneumann.org/?sampleText=2020-03-12T13%3A34%3A56.123Z&flags=Pi&selection=0%7CDate))

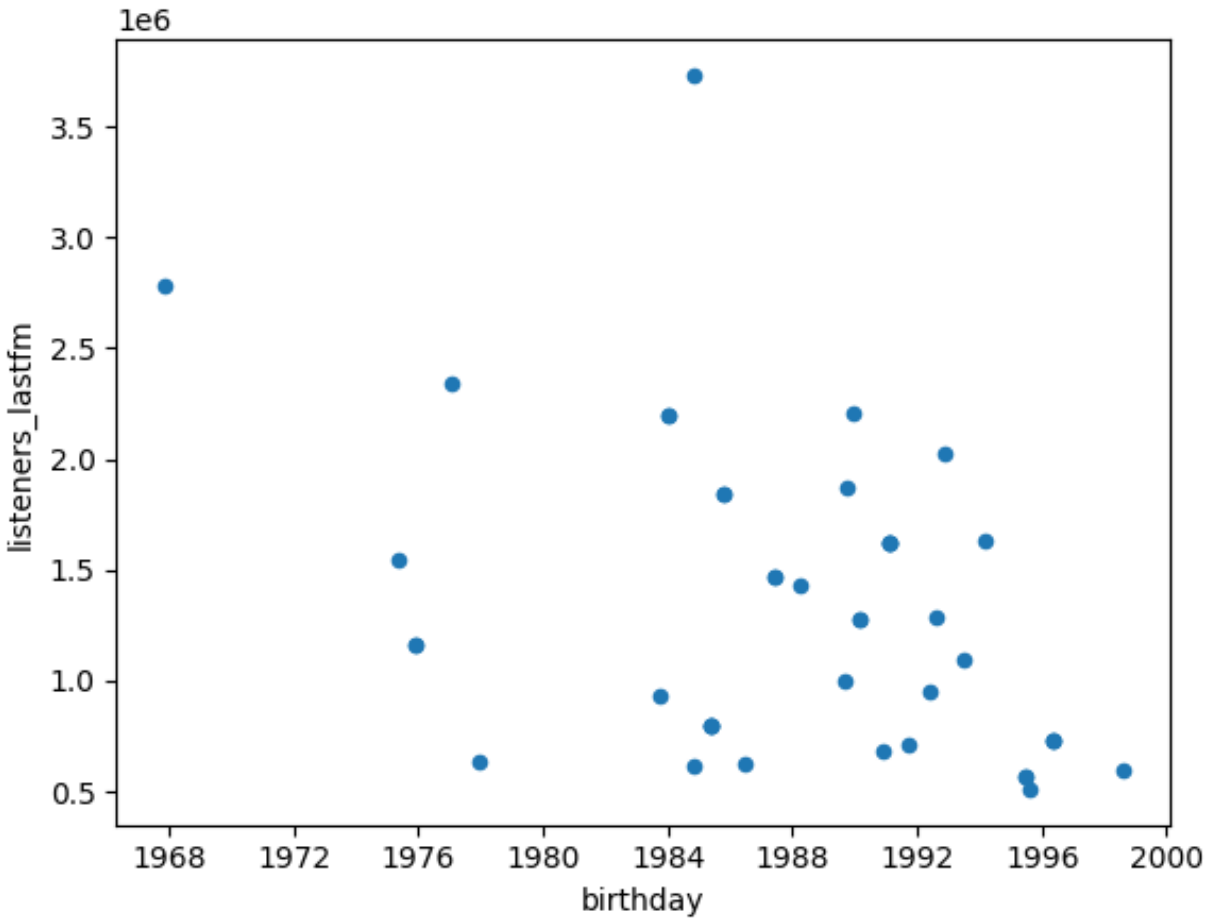
In [ ]:
```
import re
pattern = r'\d{4}-\d{2}-\d{2}'
# Find only the rows that have valid birthdays
only_bdays = tracks_df[tracks_df["birthday"].str.match(pattern)].cop
y()
# Convert to a datetime format
bdays["birthday"] = pd.to_datetime(only_bdays["birthday"])
```

Of course, we could have just ruled out our "Inconclusive" entries like so. But regex is a very useful tool to have on hand!

In [ ]:
```
only_bdays = tracks_df[tracks_df["birthday"] != "Inconclusive"]
```

In [ ]:
```
# Do a quick scatter of one variable against each other
bdays.plot.scatter("birthday", "listeners_lastfm")
```

Out[ ]: <AxesSubplot:xlabel='birthday', ylabel='listeners_lastfm'>



Your turn!