# **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:

**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

tracks\_df = pd.read\_csv('data/spotify\_2017.csv')

tracks\_df.head(2)

|  | **id** | **name** | **artists** | **danceability** | **energy** | **key** | **loudness** | **mode** | **speechiness** | **acousticness** | **...** | **liveness** | **valence** | **tempo** | **duration\_ms** | **time\_signature** | **lyrics** | **artist\_mb** | **listeners\_lastfm** | **music\_brainz\_plays** | **birthday** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 7qiZfU4dY1lWllzX7mPBI | Shape of You | Ed Sheeran | 0.825 | 0.652 | 1.0 | -3.183 | 0.0 | 0.0802 | 0.581 | ... | 0.0931 | 0.931 | 95.977 | 233713.0 | 4.0 | The club isn't the best place to find a lover\... | Ed Sheeran | 1620379.0 | 20413 | 1991-02-17 |
| **1** | 5CtI0qwDJkDQGwXD1H1cL | Despacito - Remix | Luis Fonsi | 0.694 | 0.815 | 2.0 | -4.328 | 1.0 | 0.1200 | 0.229 | ... | 0.0924 | 0.813 | 88.931 | 228827.0 | 4.0 |  | NaN | NaN | 2444 | 1978-04-15 |

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

**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']

fetch\_lyrics('The Beatles', 'Come Together')[0:100]

'Here come old flat top\nHe come grooving up slowly\nHe got joo joo eyeball\nHe one holy roller\nHe got h'

fetch\_lyrics('The Beatles', "Wouldn't it be nice")

''

💪 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 definition with:

**from** **music** **import**<SPACE><TAB>

**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.

%**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)

tracks\_df['lyrics'] = ""

%%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

print(tracks\_df.loc[0, 'lyrics'][0:150])

We're good to proceed with our whole DataFrame!

**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!

**import** **pandas** **as** **pd**

**import** **sqlite3**

conn = sqlite3.connect("data/music.sqlite")

cursor = conn.cursor()

cursor.execute("SELECT name FROM sqlite\_master WHERE type='table'")

print(cursor.fetchall())

[('artist\_info',), ('popularity',)]

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"

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!

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)

listens\_df.head(3)

|  | **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 💡

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/)

!pip install --quiet pandas-gbq

**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

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.

musicbrainz\_df.columns = ["artists", "music\_brainz\_plays"]

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

## **5️⃣ 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.

**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).

*# 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:

**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:

artists\_list = list(set(tracks\_df["artists"].tolist()))

len(artists\_list)

78

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.

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 😮‍💨)

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.

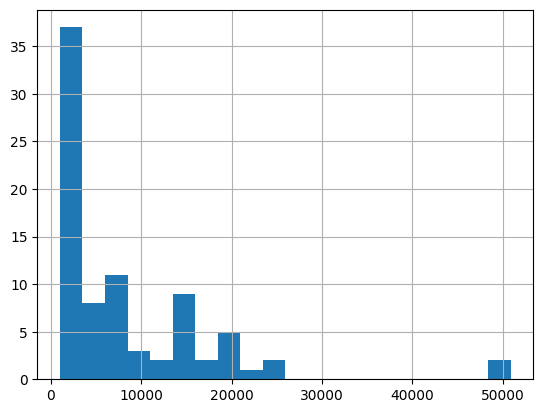
😇 For future reference - if you ever do want information from Wikipedia there is a lovely [API for Python](https://wikipedia-api.readthedocs.io/en/latest/README.html) we could have used

## **6️⃣ Quick plots with pandas**

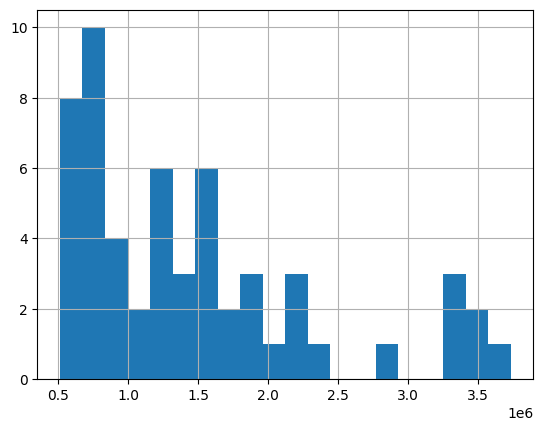
Let's use [pandas.DataFrame.hist](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.hist.html) to visualize some of the stats we've pulled together!

tracks\_df["column"].hist()

tracks\_df["music\_brainz\_plays"].hist(bins = 20);



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))

**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)].copy()

*# Convert to a datetime format*

bdays["birthday"] = pd.to\_datetime(only\_bdays["birthday"])

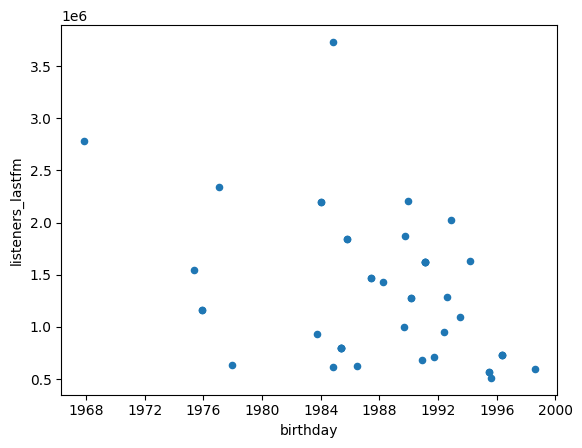
Of course, we could have just ruled out our "Inconclusive" entries like so. But regex is a very useful tool to have on hand!

only\_bdays = tracks\_df[tracks\_df["birthday"] != "Inconclusive"]

*# Do a quick scatter of one variable against each other*

bdays.plot.scatter("birthday", "listeners\_lastfm")

<AxesSubplot:xlabel='birthday', ylabel='listeners\_lastfm'>



## **Your turn!**