Week 2

Chapter 2

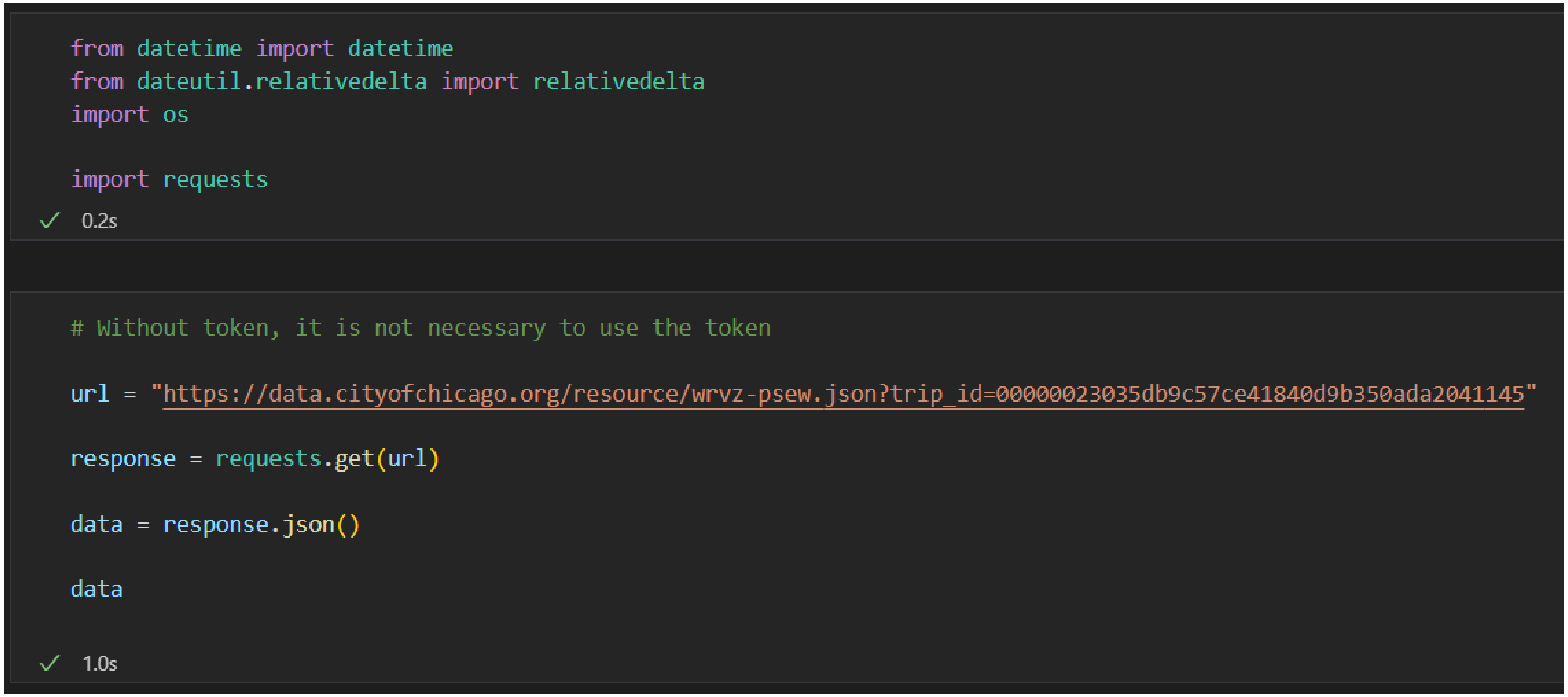
Data Engineer

Trainer: Balazs Balogh

CUBIX
INSTITUTE OF TECHNOLOGY

Get Taxi data with the API

- You don’t have to have a token to get data. No headers parameter needed in requests.get().
- Let’s try first with one trip.



```
from datetime import datetime
from dateutil.relativedelta import relativedelta
import os

import requests
```

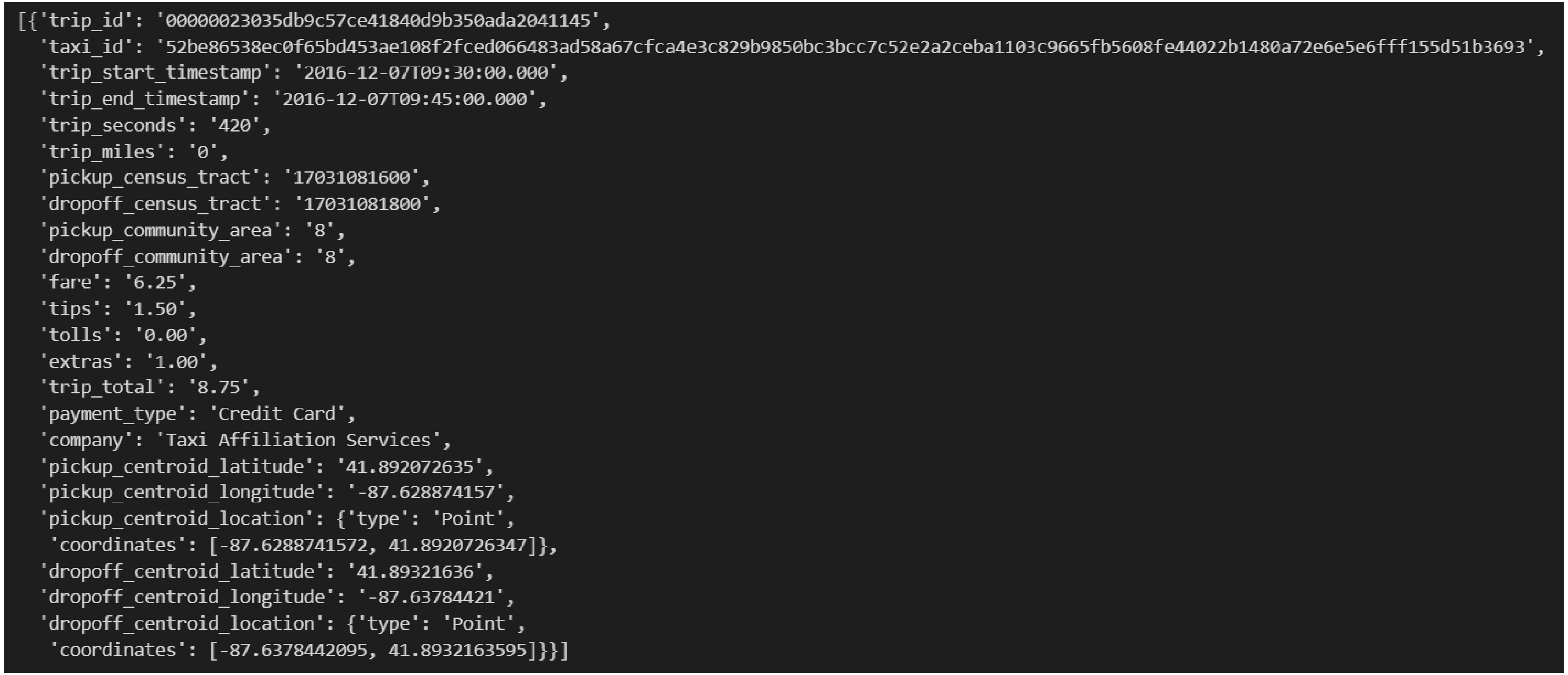
✓ 0.2s

```
# Without token, it is not necessary to use the token

url = "https://data.cityofchicago.org/resource/wrvz-psew.json?trip_id=00000023035db9c57ce41840d9b350ada2041145"

response = requests.get(url)

data = response.json()

data
```

✓ 1.0s

Get Taxi data with the API

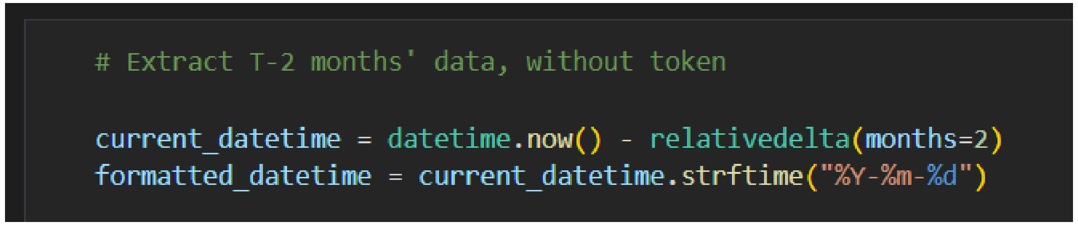
- Response:



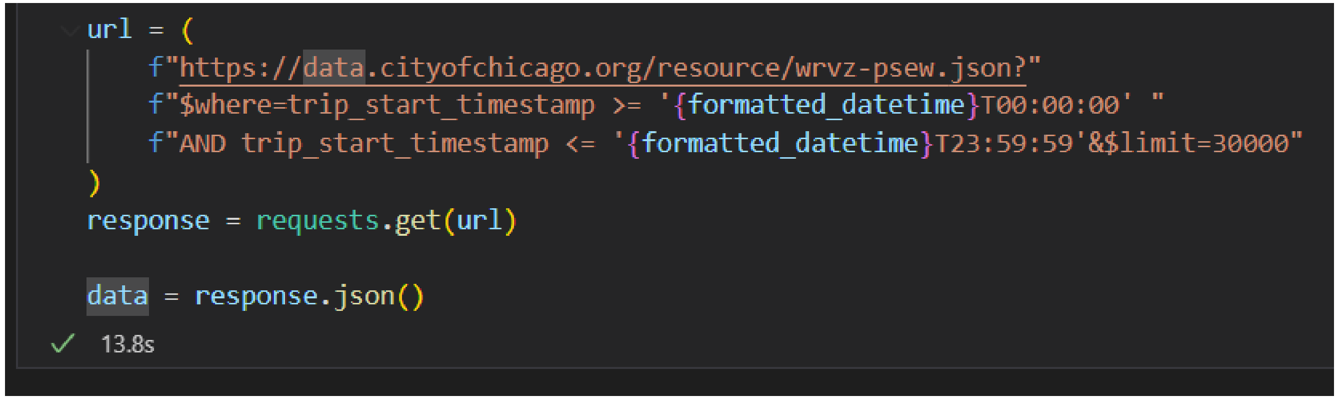
```
[{'trip_id': '00000023035db9c57ce41840d9b350ada2041145',
  'taxi_id': '52be86538ec0f65bd453ae108f2fced066483ad58a67cfca4e3c829b9850bc3bcc7c52e2a2ceba1103c9665fb5608fe44022b1480a72e6e5e6fff155d51b3693',
  'trip_start_timestamp': '2016-12-07T09:30:00.000',
  'trip_end_timestamp': '2016-12-07T09:45:00.000',
  'trip_seconds': '420',
  'trip_miles': '0',
  'pickup_census_tract': '17031081600',
  'dropoff_census_tract': '17031081800',
  'pickup_community_area': '8',
  'dropoff_community_area': '8',
  'fare': '6.25',
  'tips': '1.50',
  'tolls': '0.00',
  'extras': '1.00',
  'trip_total': '8.75',
  'payment_type': 'Credit Card',
  'company': 'Taxi Affiliation Services',
  'pickup_centroid_latitude': '41.892072635',
  'pickup_centroid_longitude': '-87.628874157',
  'pickup_centroid_location': {'type': 'Point',
   'coordinates': [-87.6288741572, 41.8920726347]},
  'dropoff_centroid_latitude': '41.89321636',
  'dropoff_centroid_longitude': '-87.63784421',
  'dropoff_centroid_location': {'type': 'Point',
   'coordinates': [-87.6378442095, 41.8932163595]}}]
```

Get Taxi data with the API

- In order to simulate daily data load, we have to go back two months in time, and download that day's data. If today is 2023-12-01, then we download 2023-10-01 data.
- One day is about 15 Mb, and around 14.000 rows.



```python
# Extract T-2 months' data, without token

current_datetime = datetime.now() - relativedelta(months=2)
formatted_datetime = current_datetime.strftime("%Y-%m-%d")
```

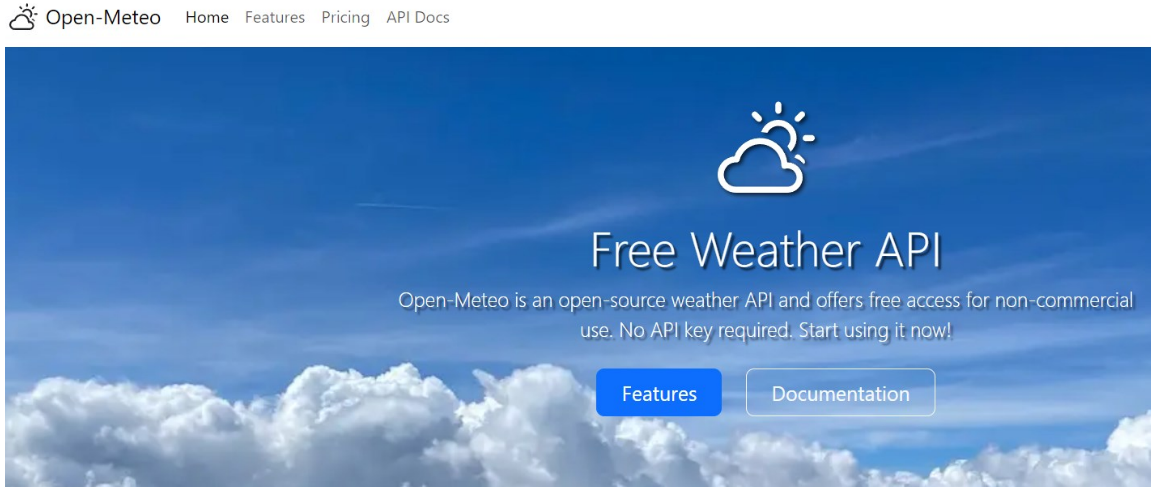


```python
url = (
    f"https://data.cityofchicago.org/resource/wrvz-psew.json?"
    f"$where=trip_start_timestamp >= '{formatted_datetime}T00:00:00' "
    f"AND trip_start_timestamp <= '{formatted_datetime}T23:59:59'&$limit=30000"
)
response = requests.get(url)

data = response.json()
```

✓ 13.8s

Get weather data from public API

- Weather data is not originally part of our initial dataset, so we have to get it from a different source.
- With weather information, you can give an additional dimension to your data, and can answer more questions like, “are there more taxi rides when it is raining?”.
- Open Meteo (https://open-meteo.com/) has a free API we can call to get historical data.
- We need historical, because we are using two months old data.

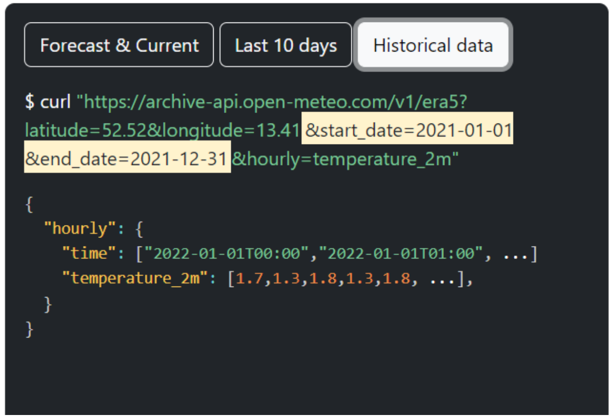


Free Weather API

Open-Meteo is an open-source weather API and offers free access for non-commercial use. No API key required. Start using it now!

Accurate Weather Forecasts for Any Location

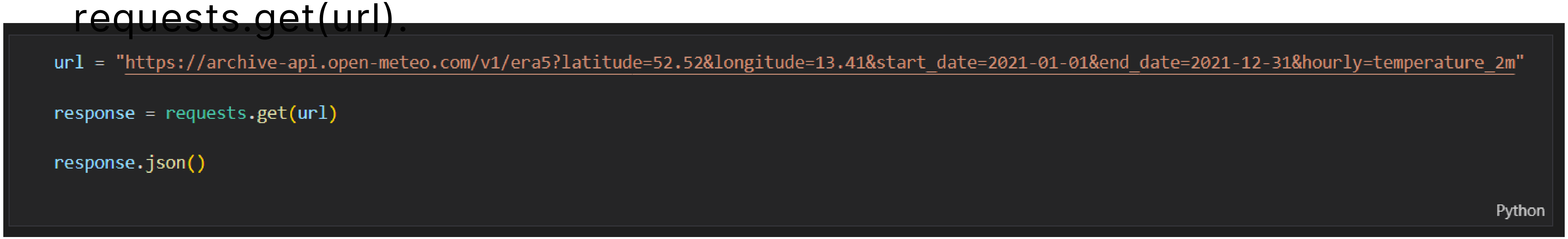
Open-Meteo partners with national weather services to bring you open data with high resolution, ranging from 1 to 11 kilometers. Our powerful APIs intelligently select the most suitable weather models for your specific location, ensuring accurate and reliable forecasts.



```
$ curl "https://archive-api.open-meteo.com/v1/era5?
latitude=52.52&longitude=13.41&start_date=2021-01-01
&end_date=2021-12-31&hourly=temperature_2m"

{
  "hourly": {
    "time": ["2022-01-01T00:00","2022-01-01T01:00", ...]
    "temperature_2m": [1.7,1.3,1.8,1.3,1.8, ...],
  }
}
```

Get weather data from public API

- Get one day’s data for testing.
- You need latitude / longitude, which is for Chicago are 52.52 / 13.41.
- Method is the same as for the Chicago API, requests.get(url).

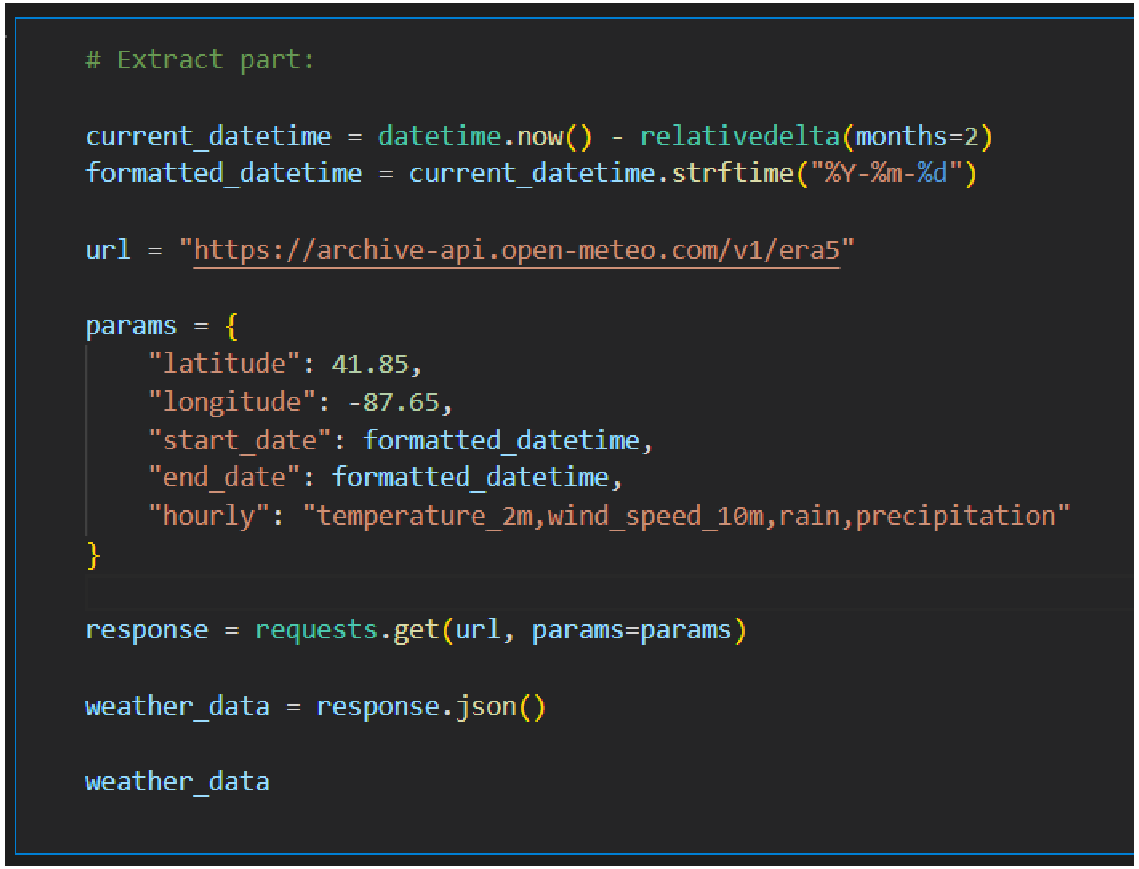


```python
url = "https://archive-api.open-meteo.com/v1/era5?latitude=52.52&longitude=13.41&start_date=2021-01-01&end_date=2021-12-31&hourly=temperature_2m"

response = requests.get(url)

response.json()
```

- This is the default request, parameters will be used.

Get weather data from public API

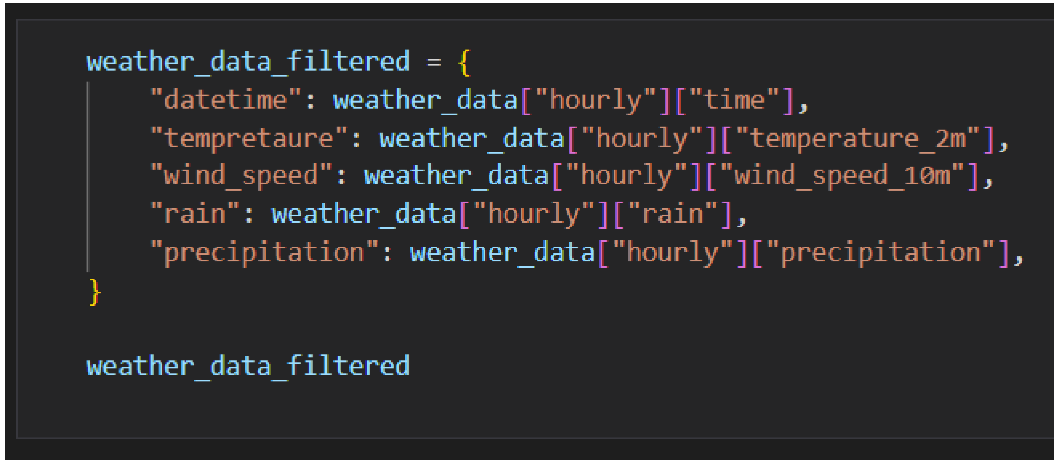
- Use the same formatted_datetime calculation as for the taxi data.
- In the params dictionary, you can specify the location, start and end date, and which information you want to retrieve. We get temperature / wind speed / rain / precipitation.



```
# Extract part:

current_datetime = datetime.now() - relativedelta(months=2)
formatted_datetime = current_datetime.strftime("%Y-%m-%d")

url = "https://archive-api.open-meteo.com/v1/era5"

params = {
    "latitude": 41.85,
    "longitude": -87.65,
    "start_date": formatted_datetime,
    "end_date": formatted_datetime,
    "hourly": "temperature_2m,wind_speed_10m,rain,precipitation"
}

response = requests.get(url, params=params)

weather_data = response.json()

weather_data
```

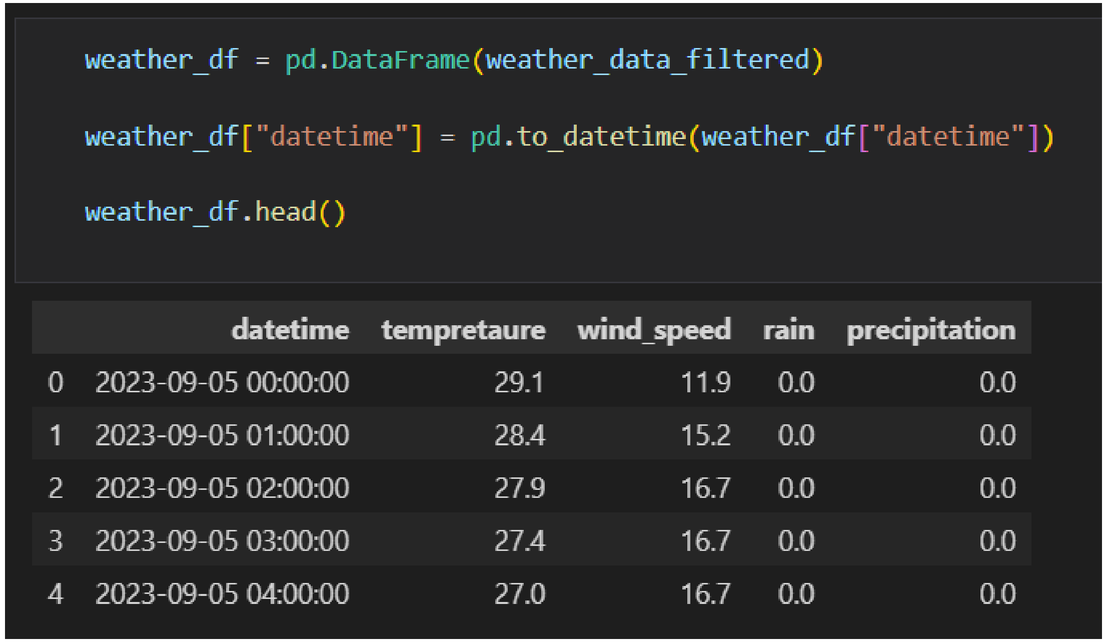
Get weather data from public API

- Format the response, create a dictionary from it with keys like “datetime”, “temperature”, “wind_speed”, “rain”, “precipitation”.



```
weather_data_filtered = {
    "datetime": weather_data["hourly"]["time"],
    "tempretaure": weather_data["hourly"]["temperature_2m"],
    "wind_speed": weather_data["hourly"]["wind_speed_10m"],
    "rain": weather_data["hourly"]["rain"],
    "precipitation": weather_data["hourly"]["precipitation"],
}

weather_data_filtered
```

- Last step is to create a DataFrame from the dictionary and save it as csv.
- Convert the “datetime” column to proper datetime format.



```
weather_df = pd.DataFrame(weather_data_filtered)

weather_df["datetime"] = pd.to_datetime(weather_df["datetime"])

weather_df.head()
```

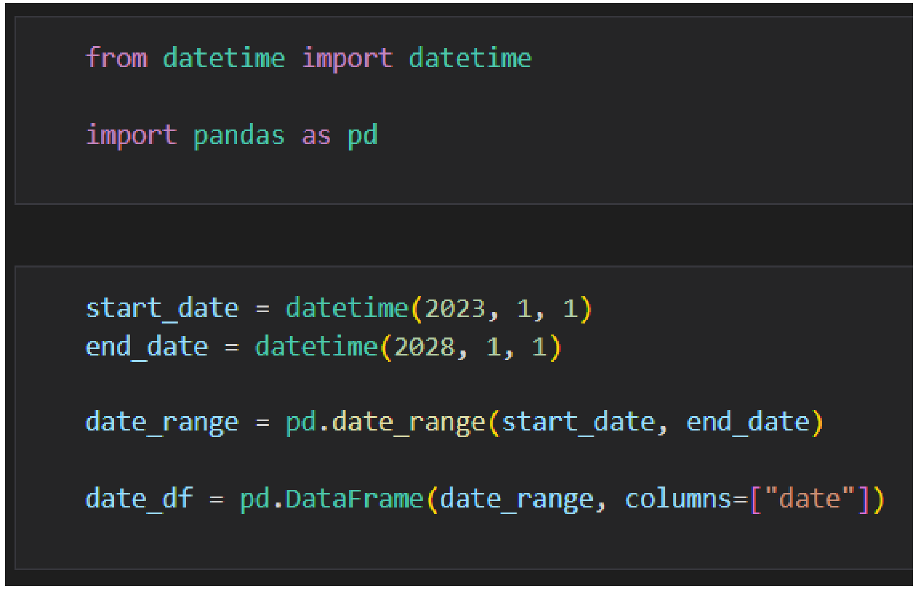
	datetime	tempretaure	wind_speed	rain	precipitation
0	2023-09-05 00:00:00	29.1	11.9	0.0	0.0
1	2023-09-05 01:00:00	28.4	15.2	0.0	0.0
2	2023-09-05 02:00:00	27.9	16.7	0.0	0.0
3	2023-09-05 03:00:00	27.4	16.7	0.0	0.0
4	2023-09-05 04:00:00	27.0	16.7	0.0	0.0

Create date dimension table

- A date dimension table is used to store information about dates, such as day, month, year, and related details (is weekend?).
- It serves as a reference table for organizing and analyzing time-based data in a data warehouse.
- It’s a tool for managing and understanding time-related aspects in a database.
- This table helps in simplifying queries, enabling efficient filtering and grouping of data based on dates, and providing a consistent structure for handling temporal information across different datasets.

Create date dimension table

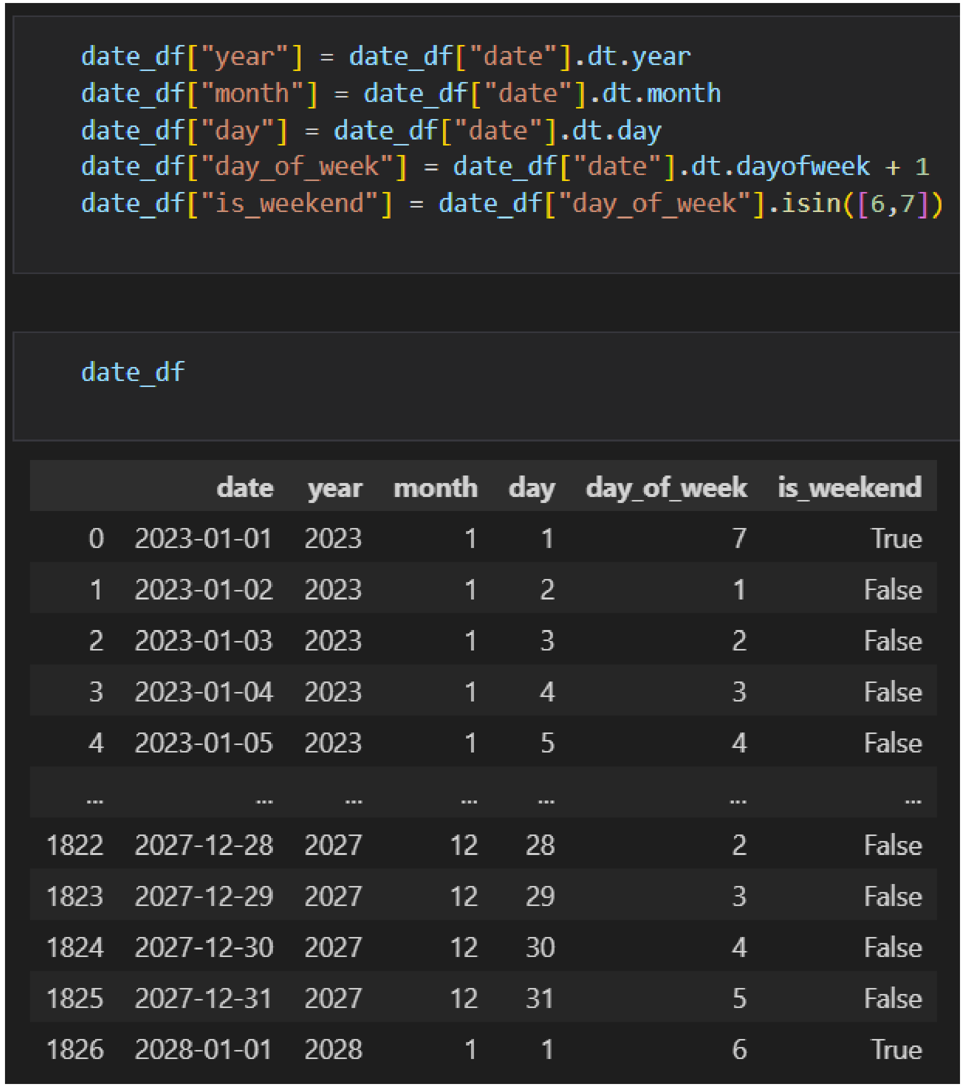
- No extra libraries needed, only the built in datetime, and pandas for creating the usual csv file from it.
- First, create dates from 2023-01-01 to 2028-01-01. Five year timeframe, should be enough.



```
from datetime import datetime

import pandas as pd
```

```
start_date = datetime(2023, 1, 1)
end_date = datetime(2028, 1, 1)

date_range = pd.date_range(start_date, end_date)

date_df = pd.DataFrame(date_range, columns=["date"])
```

Create date dimension table



```
date_df["year"] = date_df["date"].dt.year
date_df["month"] = date_df["date"].dt.month
date_df["day"] = date_df["date"].dt.day
date_df["day_of_week"] = date_df["date"].dt.dayofweek + 1
date_df["is_weekend"] = date_df["day_of_week"].isin([6,7])
```

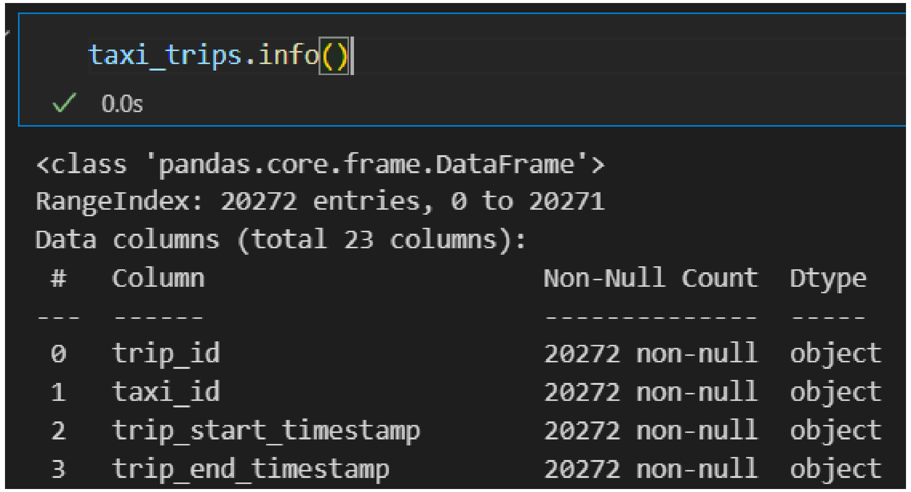
```
date_df
```

```
            date  year  month  day  day_of_week  is_weekend
   0  2023-01-01  2023      1    1            7        True
   1  2023-01-02  2023      1    2            1       False
   2  2023-01-03  2023      1    3            2       False
   3  2023-01-04  2023      1    4            3       False
   4  2023-01-05  2023      1    5            4       False
 ...         ...   ...    ...  ...          ...         ...
1822  2027-12-28  2027     12   28            2       False
1823  2027-12-29  2027     12   29            3       False
1824  2027-12-30  2027     12   30            4       False
1825  2027-12-31  2027     12   31            5       False
1826  2028-01-01  2028      1    1            6        True
```

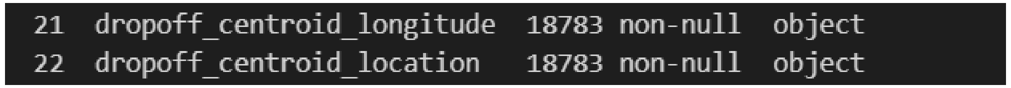
- Then create the dimensions. Year, month, day, day of week, and is weekend.
- These will help to answer questions like, which day is the busiest, or what is the difference between weekdays and weekends.
- Join it to the base data (the taxi data), and you can perform queries like above.

Exploring Taxi data

- Get one day’s data, and check it.
- It can be seen immediately, that we have NULL values.



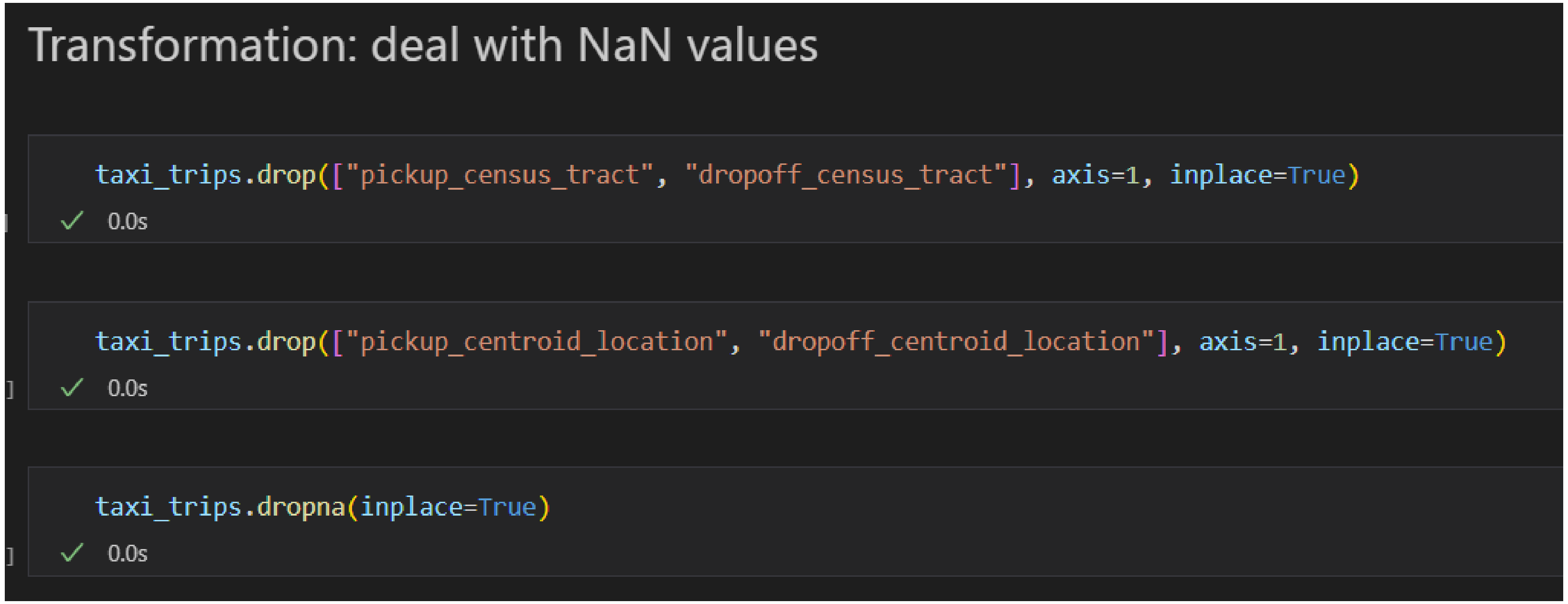
```
taxi_trips.info()
✓ 0.0s

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20272 entries, 0 to 20271
Data columns (total 23 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   trip_id                 20272 non-null  object
 1   taxi_id                 20272 non-null  object
 2   trip_start_timestamp    20272 non-null  object
 3   trip_end_timestamp      20272 non-null  object
```



```
 21  dropoff_centroid_longitude  18783 non-null  object
 22  dropoff_centroid_location   18783 non-null  object
```

Exploring Taxi data

- Just drop all the NaN (NULL) values, we won’t fill them in this time.
- For this, we will delete four columns which has the most NaN values, and then use the .dropna().



Transformation: deal with NaN values

```
taxi_trips.drop(["pickup_census_tract", "dropoff_census_tract"], axis=1, inplace=True)
```

✓ 0.0s

```
taxi_trips.drop(["pickup_centroid_location", "dropoff_centroid_location"], axis=1, inplace=True)
```

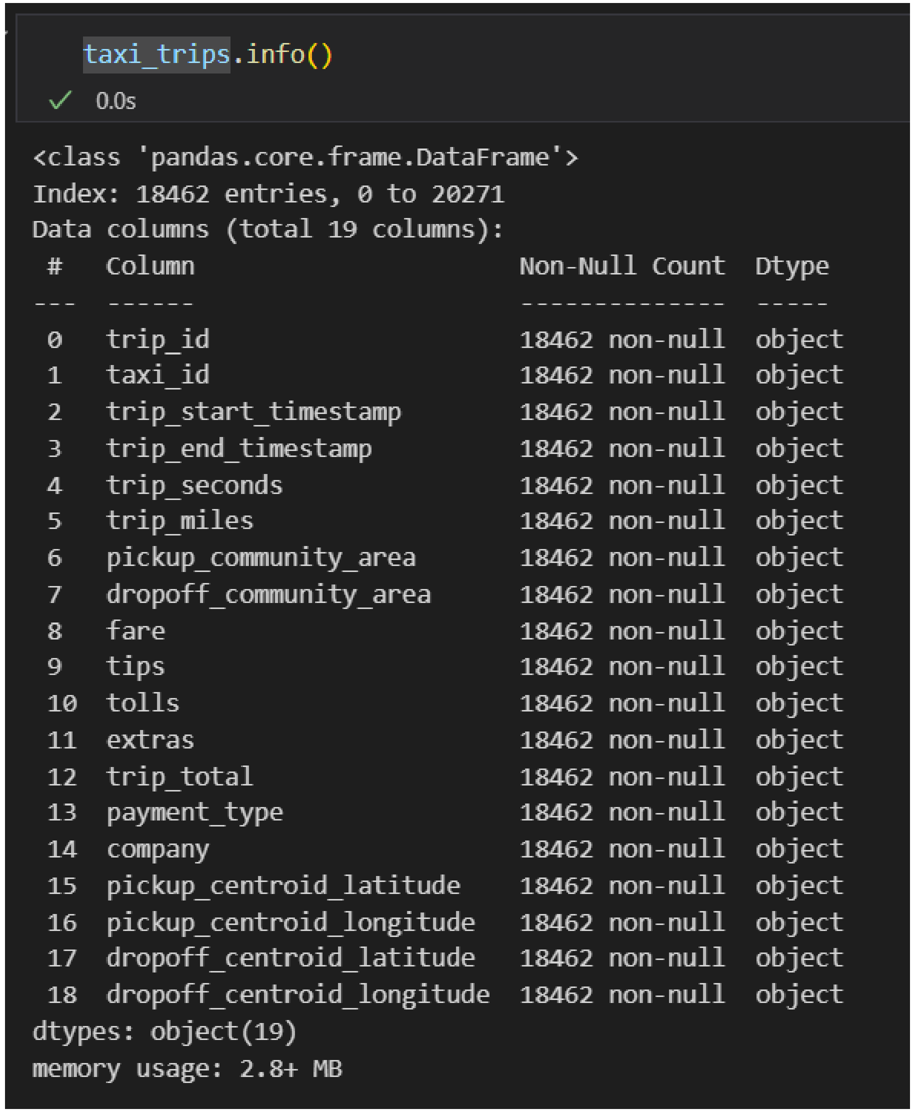
✓ 0.0s

```
taxi_trips.dropna(inplace=True)
```

✓ 0.0s

Exploring Taxi data

- Now it is consistent, zero NaN values.
- In real life maybe you can't just delete 20% of the data, so you have to fill those with some data, or just leave them as NULL, because that is also an information.



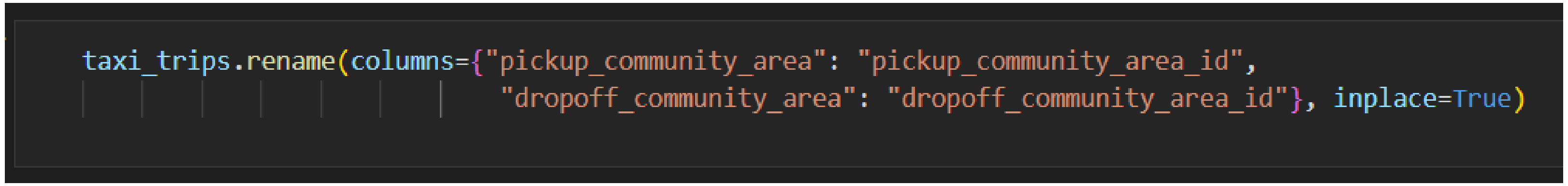
```
taxi_trips.info()
```

✓ 0.0s

```
<class 'pandas.core.frame.DataFrame'>
Index: 18462 entries, 0 to 20271
Data columns (total 19 columns):
 #   Column                     Non-Null Count  Dtype 
---  ------                     --------------  ----- 
 0   trip_id                    18462 non-null  object
 1   taxi_id                    18462 non-null  object
 2   trip_start_timestamp       18462 non-null  object
 3   trip_end_timestamp         18462 non-null  object
 4   trip_seconds               18462 non-null  object
 5   trip_miles                 18462 non-null  object
 6   pickup_community_area      18462 non-null  object
 7   dropoff_community_area     18462 non-null  object
 8   fare                       18462 non-null  object
 9   tips                       18462 non-null  object
 10  tolls                      18462 non-null  object
 11  extras                     18462 non-null  object
 12  trip_total                 18462 non-null  object
 13  payment_type               18462 non-null  object
 14  company                    18462 non-null  object
 15  pickup_centroid_latitude   18462 non-null  object
 16  pickup_centroid_longitude  18462 non-null  object
 17  dropoff_centroid_latitude  18462 non-null  object
 18  dropoff_centroid_longitude 18462 non-null  object
dtypes: object(19)
memory usage: 2.8+ MB
```

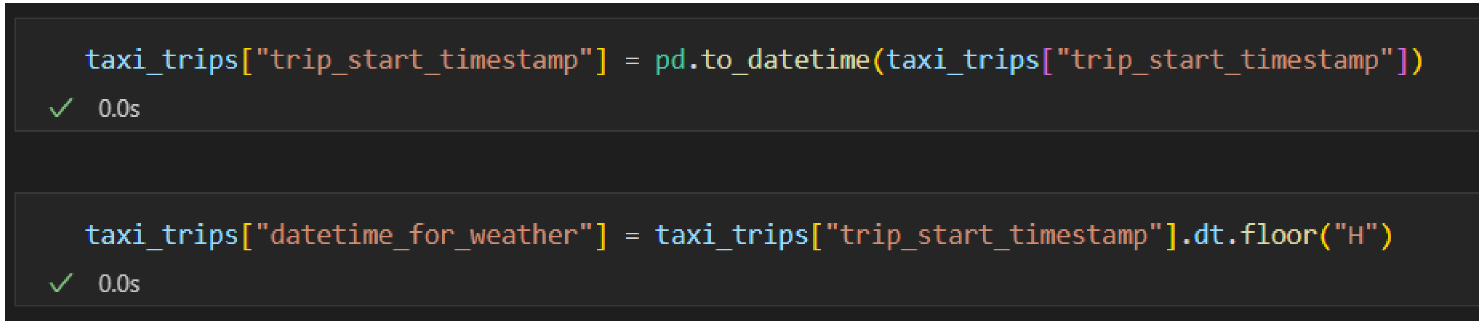
Exploring Taxi data

- Rename columns, pickup_community_area and dropoff_community_area to _id, because they are just ids, we will join the scraped names from Wikipedia.



```
taxi_trips.rename(columns={"pickup_community_area": "pickup_community_area_id",
                           "dropoff_community_area": "dropoff_community_area_id"}, inplace=True)
```

- Create a helper column for the date dimension table.
- dt.floor("H") means rounding to the hour value.

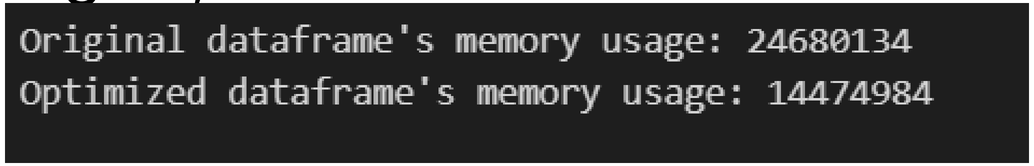


```
taxi_trips["trip_start_timestamp"] = pd.to_datetime(taxi_trips["trip_start_timestamp"])
✓ 0.0s

taxi_trips["datetime_for_weather"] = taxi_trips["trip_start_timestamp"].dt.floor("H")
✓ 0.0s
```

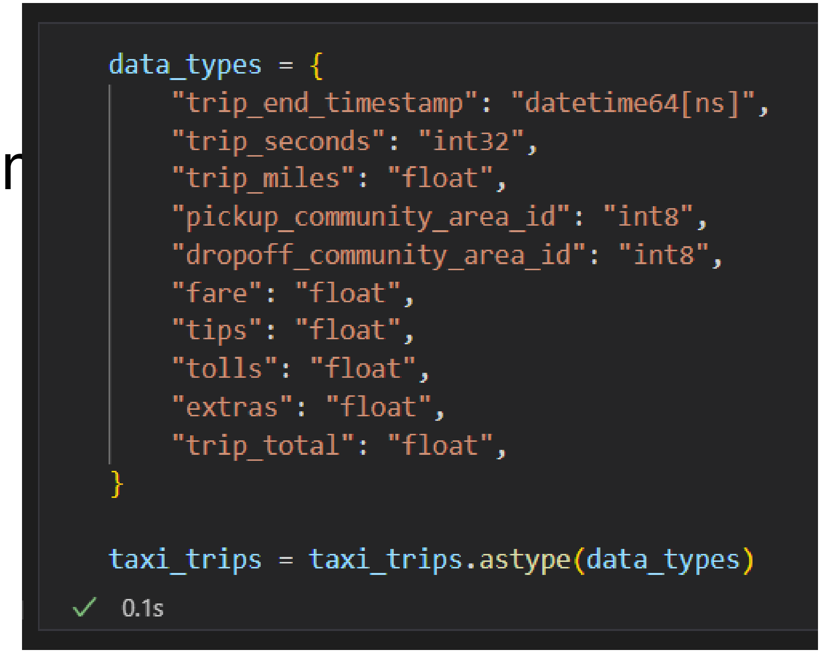
Exploring Taxi data

- Type conversions for the DataFrame.
- By default it uses around 25 Mbytes of mem

- Convert the not object columns to their respective type
- Check the memory usage again, now it uses 14



```
Original dataframe's memory usage: 24680134
Optimized dataframe's memory usage: 14474984
```

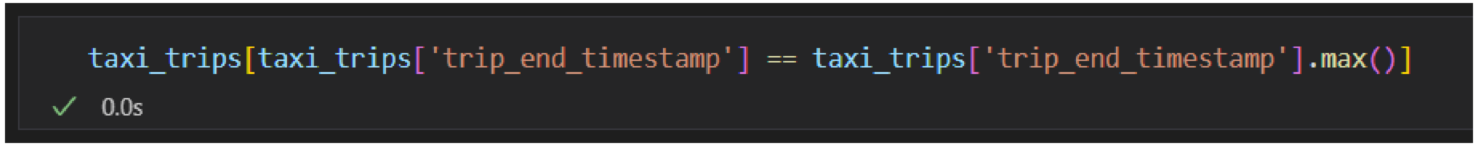


```
data_types = {
    "trip_end_timestamp": "datetime64[ns]",
    "trip_seconds": "int32",
    "trip_miles": "float",
    "pickup_community_area_id": "int8",
    "dropoff_community_area_id": "int8",
    "fare": "float",
    "tips": "float",
    "tolls": "float",
    "extras": "float",
    "trip_total": "float",
}

taxi_trips = taxi_trips.astype(data_types)
```

✓ 0.1s

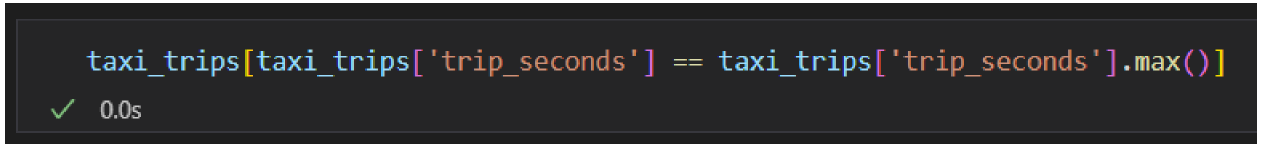
Exploring Taxi data

- Do some sanity checks to spot out strange values, or outliers.
- They are basic tests and validations to ensure that data is accurate, consistent, and meets expected standards, helping to catch errors and maintain data quality throughout the processing pipeline.
- In a real project, you will contact your end user (Data Scientist, Data Analyst etc.) and come up with ideas, how to solve these issues.



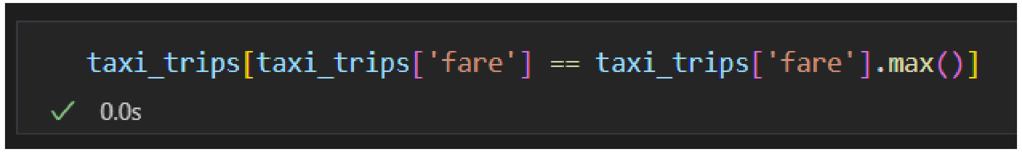
```
taxi_trips[taxi_trips['trip_end_timestamp'] == taxi_trips['trip_end_timestamp'].max()]
```

✓ 0.0s



```
taxi_trips[taxi_trips['trip_seconds'] == taxi_trips['trip_seconds'].max()]
```

✓ 0.0s



```
taxi_trips[taxi_trips['fare'] == taxi_trips['fare'].max()]
```

✓ 0.0s