Pandas IV

Raul P. Pelaez

February 17, 2026

Contents

1 Introduction

In this lesson, we will practice some more Pandas by covering:

- Data cleaning: Handling missing values, duplicates
- Getting data from the internet

Info

API: Application Programming Interface
An API is a set of rules and protocols that allow different software applications to communicate with each other. For instance, the OpenAI API allows to interact with an LLM model through the internet:

```
import requests
import json
url = "https://api.openai.com/v1/chat/completions"
openai_key = "your_openai_key"
headers = {
    "Content-Type": "application/json",
    "Authorization": openai_key
}
data = {
    "model": "gpt-4o-mini",
    "messages": [{"role": "user", "content": "Say this is a test!"}],
    "temperature": 0.7
}
response = requests.post(url, headers=headers, data=json.dumps(data))
print(response.json())
```

In the example above, we are using the requests library to send a POST request to the OpenAI API. The request includes additional data ("model", "messages", "temperature") that the API understands. The response (as JSON) is then printed to the console.

Info

API keys
Most APIs require an API key to access their services. The key is a way of authentication and authorization. It tells the API who you are and what you can do. Getting a key highly depends on the API you are using. Some are free, some are paid, some require you to register, some don't. Check the documentation of the API you are using to know how to get a key. Try searching "whatever api key".
For instance, you can get a free key for AEMET using your mail. Or for the Gemini LLM here.

Warning

This key is unique to you and should be kept secret. If you are sharing code with others, make sure to remove your API key before sharing it.

2 Data cleaning

It's wild out there, each datasets come with their own quirks and problems. If you are lucky this means some missing values, duplicates, or errors. If you are unlucky, it means all of the above and more. Lets see how to deal with some common problems.

Info

Missing values: Not Available
In Pandas terminology, we use the terms "missing", "NA" (not available), and "null" interchangeably to refer to missing values. These are represented by the `NaN` value in Pandas. What constitutes "missing" is what you intuitively think: empty strings, `None`, `NaN`, etc. See this example:

```
import pandas as pd
import numpy as np
data = pd.DataFrame([[1, 2, 3], [1, None, 6], [7, None, np.nan]],
                    columns=["A", "B", "C"])
print(data)
```

Output

```
   A    B    C
0  1  2.0  3.0
1  1  NaN  6.0
2  7  NaN  NaN
```

Info

Handling missing values

Let me be extremely practical here with some examples. Pandas normally identifies methods dealing with missing values with the `na` suffix. Lets work with this example:

```
import pandas as pd
import numpy as np
data = pd.DataFrame([[1, 2, 3], [1, None, 6], [7, None, np.nan]],
                    columns=["A", "B", "C"])
print(data)
```

Output

```
   A    B    C
0  1  2.0  3.0
1  1  NaN  6.0
2  7  NaN  NaN
```

Here are some common methods to handle missing values:

```
# Drop rows with missing values
print("--------------data.dropna()--------------")
print(data.dropna())
# Drop columns with missing values
print("--------------data.dropna(axis=1)--------")
print(data.dropna(axis=1))
# Fill missing values with a specific value
print("--------------data.fillna(-10)------------")
print(data.fillna(-10))
```

Output

```
--------------data.dropna()--------------
   A    B    C
0  1  2.0  3.0
--------------data.dropna(axis=1)--------
   A
0  1
1  1
2  7
--------------data.fillna(-10)------------
   A     B     C
0  1   2.0   3.0
1  1 -10.0   6.0
2  7 -10.0 -10.0
```

As usual, these methods offer a plethora of options to customize their behavior. Check their docs! Additionally, all the `read_` functions in Pandas have parameters like `na_values`, which specifies what should be considered missing (say, the string "N/A").

Info

Duplicated data

In Pandas, duplicate data refers to rows that have identical values across some or all columns. Duplicates might represent genuine repeated observations or data entry errors that need attention. Here's an example to illustrate:

```
import pandas as pd
data = pd.DataFrame([
    ['John', 25, 'NY'],
    ['Mary', 30, 'CA'],
    ['John', 25, 'NY'],  # Duplicate row
    ['Peter', 35, 'TX']
], columns=['Name', 'Age', 'State'])
print(data)
```

Output

```
    Name  Age State
0   John   25    NY
1   Mary   30    CA
2   John   25    NY
3  Peter   35    TX
```

Info

Handling duplicates

Let's explore practical methods for working with duplicates in Pandas. The main methods have the prefix `duplicate` in their names. Using our example data:

```
# Check which rows are duplicates
print("----------data.duplicated()----------")
print(data.duplicated())  # Returns boolean Series

# Find duplicate rows
print("----------data[data.duplicated()]----------")
print(data[data.duplicated()])

# Remove duplicates and keep first occurrence
print("----------data.drop_duplicates()----------")
print(data.drop_duplicates())

# Remove duplicates and keep last occurrence
print("----------data.drop_duplicates(keep='last')----------")
print(data.drop_duplicates(keep='last'))
```

Output

```
----------data.duplicated()----------
0    False
1    False
2     True
3    False
dtype: bool
----------data[data.duplicated()]----------
   Name  Age State
2  John   25    NY
----------data.drop_duplicates()----------
    Name  Age State
0   John   25    NY
1   Mary   30    CA
3  Peter   35    TX
----------data.drop_duplicates(keep='last')----------
    Name  Age State
1   Mary   30    CA
2   John   25    NY
3  Peter   35    TX
```

These methods offer additional parameters for fine-tuning:

- `subset` checks duplicates in specific columns: `data.drop_duplicates(subset=['Name'])`
- `keep` parameter accepts 'first', 'last', or False (drops all duplicates)

Remember that `duplicated()` and `drop_duplicates()` consider all columns by default, but you can specify which columns to consider using the `subset` parameter.

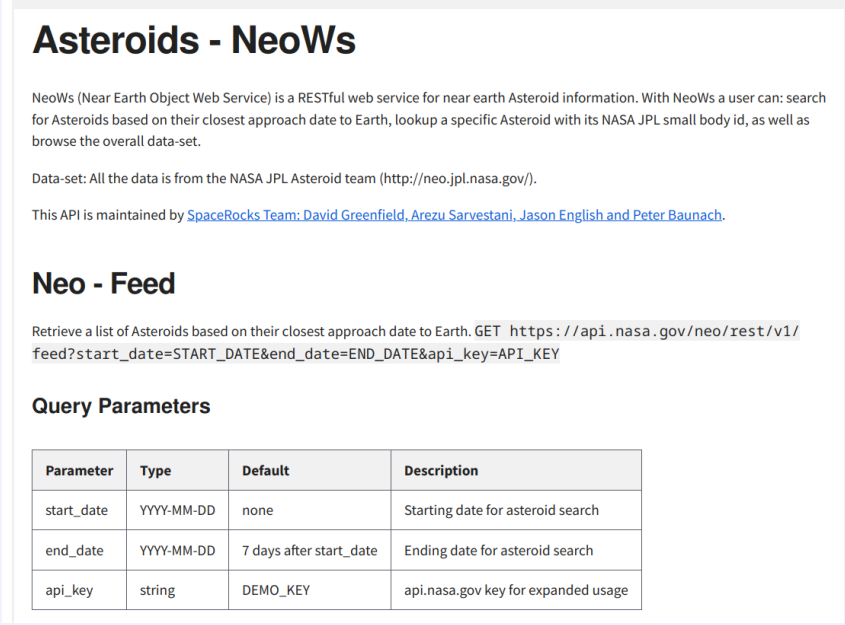
3 Getting data from the internet

Info

Requests

The Python `requests` library is a powerful tool for working with HTTP requests. It allows you to send GET and POST requests to web servers and receive responses. You can use it to download webpages, APIs, and other online resources.

ChatGPT is good at translating between requests frameworks. For instance, this is a screenshot of the documentation for a NASA API:



Asteroids - NeoWs

NeoWs (Near Earth Object Web Service) is a RESTful web service for near earth Asteroid information. With NeoWs a user can: search for Asteroids based on their closest approach date to Earth, lookup a specific Asteroid with its NASA JPL small body id, as well as browse the overall data-set.

Data-set: All the data is from the NASA JPL Asteroid team (http://neo.jpl.nasa.gov/).

This API is maintained by SpaceRocks Team: David Greenfield, Arezu Sarvestani, Jason English and Peter Baunach.

Neo - Feed

Retrieve a list of Asteroids based on their closest approach date to Earth. `GET https://api.nasa.gov/neo/rest/v1/feed?start_date=START_DATE&end_date=END_DATE&api_key=API_KEY`

Query Parameters

Parameter	Type	Default	Description
start_date	YYYY-MM-DD	none	Starting date for asteroid search
end_date	YYYY-MM-DD	7 days after start_date	Ending date for asteroid search
api_key	string	DEMO_KEY	api.nasa.gov key for expanded usage

Try copy pasting that and prompt ChatGPT to: "Translate into a Python requests query". You will get something like this:

```python
import requests
response = requests.get(
    "https://api.nasa.gov/neo/rest/v1/feed",
    params={"start_date": "2015-09-07",
            "end_date": "2015-09-08",
            "api_key": "DEMO_KEY"}
)
response.raise_for_status()  # Raise an exception for HTTP errors
print(response.json())
```

Those lines cover 99% of the requests you will ever need to make. The first argument is the URL, the second is a dictionary with the parameters. The response is a JSON object that you can parse with the `json` library, or directly transform to a Pandas `DataFrame`.

Info

GET and POST

`GET` and `POST` are two common HTTP methods used to send requests to web servers. `GET` requests are used to retrieve data from a server (i.e. download a webpage given a url), while `POST` requests are used to send data to a server. The requests library in Python supports both methods.

Using one or the other depends on the API you are interacting with. For instance, the OpenAI API uses `POST` requests to send data to the server, while the Wikipedia API uses `GET` requests to retrieve data. `GET` requests append parameters to the url, while `POST` requests can send an arbitrary amount of data to the server. Typically, APIs that require a "small" amount of information (say a weather station that requires a date) use `GET` requests, while APIs that require a "large" amount of information (say a chatbot that requires a conversation) use `POST` requests.

Advanced

User agents: hiding who you are
Some websites block requests from bots and scrapers. They pull all kinds of tricks to detect them, like asking for a Captcha. Many times, though, webpages use the "user agent", a string that identifies the browser and operating system of the user. Many times you can get away with simply changing this string to match that of a browser. For instance, this is how you can do it with requests:

```
import requests
url = "https://some_website.com"
headers = {
"User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) ..."
}
response = requests.get(url, headers=headers)
print(response.text)
```

Look online to find out the user agent of the browser you want to mimic.

Info

Pandas understands URLs
You can just pass a URL to the `read_csv` function and Pandas will download the file for you. For instance:

```
import pandas as pd
base = "https://huggingface.co/datasets/shwetha729/quantum-machine-learning/"
url = base+"raw/main/Quantum-wind%20Google%20Scholar%20Article%20Information.csv"
df = pd.read_csv(url)
print(df.head())
```

Output

```
                     Title  ...                                Related_articles_Link
0        Quantum plasmonics  ...  https://scholar.google.com/scholar?q=related:z...
1    Quantum chromodynamics  ...  https://scholar.google.com/scholar?q=related:s...
2     Quantum hadrodynamics  ...  https://scholar.google.com/scholar?q=related:m...
3  [BOOK][B] Quantum theory  ...  https://scholar.google.com/scholar?q=related:m...
4         Quantum computing  ...  https://scholar.google.com/scholar?q=related:a...

[5 rows x 7 columns]
```

3.1 Read a webpage into a DataFrame

Info

Web scraping
Web scraping is the process of extracting information from websites. This can be done manually, but it's often automated using a web scraper.

Info

HTML tables

HTML tables are a common way to display data on a webpage. They are structured using the `<table>` tag, with rows defined by the `<tr>` tag and cells by the `<td>` tag. Pandas can read HTML tables directly into a DataFrame using the `read_html()` function.

```
import pandas as pd
import requests
from io import StringIO
url = "https://en.wikipedia.org/wiki/List_of_countries_by_population_(United_Nations)"
r = requests.get(url, headers={'User-Agent': 'Mozilla/5.0'})
tables = pd.read_html(StringIO(r.text))
df = tables[0]
print(df.info())
```

Output

```
<class 'pandas.DataFrame'>
RangeIndex: 238 entries, 0 to 237
Data columns (total 6 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   Country or territory           238 non-null    str
 1   Population (1 July 2022)       238 non-null    int64
 2   Population (1 July 2023)       238 non-null    int64
 3   Change (%)                     238 non-null    str
 4   UN continental region[1]       238 non-null    str
 5   UN statistical subregion[1]    238 non-null    str
dtypes: int64(2), str(4)
memory usage: 11.3 KB
None
```

Running this code requires conda installing `lxml` and `html5lib`.

4 Exercises

4.1 Cars

Goal

Lets work on a rather noisy dataset today. It contains data for most car models ever created. Download `cars.csv` from BlackBoard. In particular, the goal is to obtain a clean dataset that contains just the following columns:

- `model_name` (str)
- `horsepower` (float)
- `top_speed` (float), in km/h
- `year` (int), the year the car was commercialized

Plot, for most models possible, the horse power vs top speed. What can you infer from the plot?

Milestone

Load the dataset and do some exploratory analysis to identify the interesting columns. Save only those to a new DataFrame.

Milestone

Clean the dataset. Handle missing values, duplicates, and any other problem you find.

hint

- As usual, supposedly numerical data comes bundled with a bunch of text. You will need to use regular expressions to extract the numbers.
- The `str.extract` method is your friend here.
- The `dropna`, `isna`, etc methods and their friends play along well with boolean indexing.

Milestone

Plot the horsepower vs top speed for the cars. The image is probably rather noisy, you will have to iterate a bit to get rid of all the spurious data. How does the plot change if you only consider cars from a specific time range?

hint

You can use the Pandas `plot` method for this, or Matplotlib if you prefer.

Advanced milestone

Devise a model that approximates the curve you see. Write a function with this signature:

```
def estimate_top_speed(horsepower: float) -> float:
    """ Estimate the top speed of a car given its horsepower

      Args:
          horsepower (float): The horsepower of the car

      Returns:
          float: The estimated top speed of the car in km/h
    """
```

hint

- You need to solve where the engine's power equals the power required to overcome aerodynamic drag.

The key equation is:

$$\text{Power} = \frac{1}{2}\rho A C_d v^3$$

Where:

- Power is the power of the engine
- ρ is the air density
- A is the frontal area of the car
- C_d is the drag coefficient
- v is the speed of the car

Be careful with the units!

4.2 Movies

Goal

Explore the MovieLens 1M Dataset, hosted here. This dataset contains movie ratings data collected from users of a webpage in the late 1990s. Process this dataset and answer the following questions:

- Is there a bias in the ratings when considering the occupation of the user?
- List the top 10 movies for which the ratings are most polarized between genders.
- What is the average rating for each genre?

hint

- Check the README in the provided url, which contains a thorough description of the dataset.
- This dataset does not follow any standard file format (like CSV), use `pd.read_table` to load it. You will need to customize the separator and the column names. Set "python" as the engine.
- You will need to merge the three tables to get the full dataset (use `merge`).
- As per usual, the dataset contains missing entries, duplicates, typos...

4.3 Real time data

Goal

Find some API about something you are interested in (searching for APIs is a skill in itself). Use the requests library to get some data from it, transform it into a Pandas DataFrame, and do some exploratory analysis.
Some ideas:

- Madrid Open Data
- NASA API
- CoinGecko API (warning, super expensive if not in the demo plan)
- AEMET API (Spain's weather data)
- Idealista API (Spain's real state data, free but requires registration and takes a while to get the key)
- Spotify API

Advanced milestone

Using LLMs

Enhance your application with an LLM. For instance, you can use the Gemini API to fill some missing data, or translate the data to another language. If you want some inspiration, let me give you an actual example. This code will request data from the Madrid OpenData API about "official" events taking place in the city:

```
def get_madrid_data(catalogue_entry, options=None):
    url = f"https://datos.madrid.es/api/3/action/package_show?id={catalogue_entry}"
    pkg = requests.get(url, timeout=30).json()
    json_res = next(
        (r for r in pkg["result"]["resources"] if r.get("format") == "JSON"), None
    )
    if not json_res:
        raise RuntimeError("No JSON resource found")

    url = json_res["url"]
    headers = {"Accept": "application/json"}
    response = requests.get(url, headers=headers, allow_redirects=True, params=options)
    response.raise_for_status()
    return response

response = get_madrid_data("300107-0-agenda-actividades-eventos")
data = pd.json_normalize(pd.json_normalize(response.json())["@graph"][0])
data.to_csv("madrid_events.csv") # Save the data, as the API is rate-limited
```

One of the fields is a description of the event, but it is in Spanish. You can use the Gemini API to translate it to English, write a function with this signature:

```
def translate_event_description(event_description: str) -> str:
""" Translate the event description to English

        Args:
        event_description (str): The description of the event in Spanish

        Returns:
        str: The description of the event in English
"""
```

Advanced milestone

This one I honestly could not crack myself. This webpage shows the current capacity of a climbing center I frequent: `https://sputnikclimbing.deporsite.net/aforo-alcobendas` By using `requests` one can download the webpage, but if you do this, you will find that the capacity is nowhere to be found. At some point, this website changed their server so that the capacity is loaded dynamically using Javascript. This means that the capacity is not in the HTML of the webpage, but is loaded by some JavaScript code after the page is loaded. `requests` does not execute JavaScript, so it cannot get the capacity.

If you are in for a challenge and want to learn a bit about web scraping, explore the `selenium` library and see if you can write a function that will return the current capacity of the climbing center.

- Make it so that your program prints the current capacity and the time of the request.
- Make it so that at every request, the program appends the capacity to a CSV file.