Pandas I

Raul P. Pelaez

January 27, 2026

Contents

1 Introduction

Pandas is a high-level data manipulation tool. Its key data structure is called the `DataFrame`, which allow us to store and manipulate tabular data (we can think of the rows as different "observations" and the columns as the variables).

Advice

Intuitively, pandas `DataFrames` can be thought of as a way to hold Excel spreadsheets data in a Python object.

Let me give you an example of tabular data:

	city	country	population
0	Tokyo	Japan	37732000
1	Jakarta	Indonesia	33756000
2	Delhi	India	32226000
3	Guangzhou	China	26940000
4	Mumbai	India	24973000

And another one:

	Model	Top Speed	From year
0	AC Ace	140 mph (225 km/h)	1993
2	AC Cobra MkIII	140 mph (225 km/h)	1963
3	AC Cobra 378 MkIV	NaN	1963
4	AC Cobra MkII	141 mph (227 km/h)	1963
...	...	...	...
6939	Zenvo TSR-S	202 mph (325 km/h)	2018
6940	Zenvo ST1	233 mph (375 km/h)	2009

Today, we will cover the basics of Pandas, starting with its two main data structures: `Series` and `DataFrame`. We will also cover the basic operations that can be performed on these data structures, reading data into pandas and basic data exploring.

Insight

The name "Pandas" is derived from the term "Panel Data", an econometrics term for multidimensional structured data sets.

Advice

What is important about tabular data?
Data surrounds us, permeating nearly every aspect of modern life. From the mundane—like the sensor data in our smartphones—to the monumental — like the vast datasets used in scientific research and global commerce. Its ubiquity presents both immense opportunities and significant challenges.
Examples of tabular data include the history of a stock price, meteorological records, geographical data, and the results of a scientific experiment. In all these cases, the data is organized in rows and columns, with each row representing a single observation and each column representing a different variable.

Info

Importing Pandas
Pandas is not part of the Python standard library, so you will need to install it using pip or conda. Activate your conda environment and run `conda install pandas` or `pip install pandas` if necessary.

```
import pandas as pd
```

Advice

It is a common practice to import pandas as pd. Similar as to how we import numpy as np.

Info

Basic Objects: Series and DataFrame
Fundamentally, Pandas has two main objects that we will be using:

- **Series** are one-dimensional labeled arrays capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). We can think of a Pandas Series as a single "column" of a table.
- A **Dataframe** is a 2-dimensional labeled data structure with columns of potentially different types. We can think of it like a spreadsheet or a dict of Series objects.

2 Series

In a `Series` object, the axis labels are collectively referred to as the index. The basic method to create a `Series` is to call:

```
import pandas as pd
series = pd.Series(data, index=index_name)
```

Here, data can be any one-dimensional iterable object. For example:

- A Python dict or list
- A scalar value
- A numpy array

If data is a numpy array, index must be the same length as data. If no index is passed, one will be created having values `[0, 1, ..., len(data) - 1]`.

```
import pandas as pd
import numpy as np
data = np.random.randn(5)
series = pd.Series(data, name='my_series', index=['a', 'b', 'c', 'd', 'e'])
print(series)
```

Output

```
a    0.854112
b   -0.418485
c    0.796935
d   -0.047201
e   -0.182680
Name: my_series, dtype: float64
```

Series objects have various attributes and methods for data manipulation and analysis. For example, we can access elements using indexing:

```
print(series.iloc[0])  # Access the first element
print(series['a'])  # Access the element with index 'a'
```

Output

```
0.8541118191170816
0.8541118191170816
```

And we can perform various operations, like calculating the mean:

```
print(series.mean()) # Calculate the mean of the Series
```

Output

```
0.20053611320091175
```

Advice

In many aspects, `Series` can behave as a numpy array, as it follows the Iterable interface. This means that it can be passed to functions that expect a numpy array.
For instance, we can iterate over a Series using a for loop, just like we would with a list or a numpy array.

```
for element in series:
    print(element)
```

Output

```
0.8541118191170816
-0.4184849510172546
0.7969348860917502
-0.04720138478336851
-0.18267980340364995
```

Advanced

Python Data Model
Python's consistency is one of its greatest strengths. Experienced Python programmers can often correctly predict the behavior or existence of unfamiliar features.
The so-called Python Data Model is one of the reasons why. We know that `Series` is a sort of container, so we can expect it to behave like all the other ones. For instance, we can use `len(series)` to get the length of a `Series`, and `series[0]` to access its first element.

For Pandas library implementors, adhering to the Data Model entails implementing a few specially-named methods in a class definition. For instance, a class (such as Series) will be Iterable if it implements the `__iter__` method, and it will be Sized if it implements the `__len__` method. With a few others, mainly `__getitem__`, `__setitem__`, `__delitem__`, and `__contains__`, we can make our class into a Collection, like a Python list or a numpy array.

Python takes care of looking for the correct method to call in each case. For instance, the `for` loop will try to call `__iter__` on the object, and the `len()` function will call `__len__`. This is why we can use these functions on a `Series` object.

Info

The index of a Series is used to label and identify each element of the underlying data. We can access the index of a Series with the .index attribute:

```
import pandas as pd
cities = ['Kolkata', 'Chicago', 'Toronto', 'Lisbon']
populations = [14.85, 2.71, 2.93, 0.51]
city_series = pd.Series(populations, index=cities)
print(city_series.index)
```

Output

```
Index(['Kolkata', 'Chicago', 'Toronto', 'Lisbon'], dtype='object')
```

3 DataFrame

DataFrames are essentially a dictionary of columns. As such, every column label must be unique. Each column corresponds to a Series, and different columns can be of different type. Rows are identified by an index, shared by all the columns (and may be non-unique).

Pandas is quite permissive with what can be used to create a DataFrame, including (but not limited to):

- A dictionary of 1D ndarrays, lists, dicts, or Series
- A 2-D numpy array
- A Series
- A CSV file
- Another DataFrame

In general, if it makes intuitive sense to construct a DataFrame with a certain data structure, Pandas probably supports it.

Example

```
import pandas as pd
# A dictionary of lists
data = {'col1': [1, 2], 'col2': [4, 5], 'col3': [7, 8]}
df1 = pd.DataFrame(data)
print(f"{df1=}")
# Optionally set the index
df2 = pd.DataFrame(data, index=['a', 'b'])
print(f"{df2=}")
```

```python
# A 2-D numpy array
import numpy as np
data = np.array([[1, 2], [4, 5], [7, 8]])
df3 = pd.DataFrame(data, columns=['col1', 'col2'])
print(f"{df3=}")
# A list of Series
data = [pd.Series([1, 2]), pd.Series([4, 5])]
df4 = pd.DataFrame(data, index=['a', 'b'])
print(f"{df4=}")
```

Output

```
df1=   col1  col2  col3
0     1     4     7
1     2     5     8
df2=   col1  col2  col3
a     1     4     7
b     2     5     8
df3=   col1  col2
0     1     2
1     4     5
2     7     8
df4=   0  1
a  1  2
b  4  5
```

We will see how to read from/to files in a future section.

Advice

Missing entries

Pandas will deal with missing data seamlessly. Missing data is identified as `NaN`. Functions dealing with them usually include the suffix `na` in them (e.g. `dropna` to remove rows with missing entries).

Example

```python
import pandas as pd
data = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}, {'b': 3}]
df = pd.DataFrame(data)
print(f"df=\n{df}")
print(f"df.dropna()=\n{df.dropna()}")
print(f"df.fillna(-1)=\n{df.fillna(-1)}")
```

Output

```
df=
     a   b     c
0  1.0   2   NaN
1  5.0  10  20.0
2  NaN   3   NaN
df.dropna()=
     a   b     c
1  5.0  10  20.0
df.fillna(-1)=
     a   b     c
0  1.0   2  -1.0
1  5.0  10  20.0
2 -1.0   3  -1.0
```

3.1 Reading data into Pandas

Besides creating DataFrames from scratch, we can also read data from various sources. The most common ones are CSV files, Excel files, JSON files, and SQL databases.

Example

```
# Assume we have a CSV file named 'data.csv'
df = pd.read_csv('data.csv')
# Assume we have an Excel file named 'data.xlsx'
df = pd.read_excel('data.xlsx')
# Can you guess how one reads a .json file?
```

Advanced

Everything is a "file"
Pandas is lenient with what it considers a file (this follows from the UNIX philosophy of "everything is a file"). For instance, we can read from a URL, a compressed file, or even a file-like object (like a BytesIO object).

Example

```
import pandas as pd
base_url = 'https://raw.githubusercontent.com/pandas-dev/pandas/refs/heads/main'
url = base_url+'/pandas/tests/io/data/csv/iris.csv'
df = pd.read_csv(url)
print(df.head(2))
```

Output

```
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name
0          5.1         3.5          1.4         0.2  Iris-setosa
1          4.9         3.0          1.4         0.2  Iris-setosa
```

Pandas uses the `:` prefix to differentiate between data "providers". If not present, the prefix is assumed to be `file:`. For instance, we can use `hf:` to read data from the Huggingface datasets library.

```
import pandas as pd
df = pd.read_csv("hf://datasets/scikit-learn/iris/Iris.csv")
```

Will automatically download and load the dataset hosted in this url. You need to install `huggingface_hub` to use this feature. You can install it with `pip install huggingface_hub`.

Advice

Real life datasets are noisy and messy. It is common to find missing values, inconsistent data types, corrupted data and non-standard formats. We will explore data cleaning and preprocessing in future lessons, but for now, check the documentation for `read_csv`. This function can be customized to, for instance, skip some lines of the file, change the item separators, or handle missing values (the amount of arguments is staggering).

3.2 Saving data to files

Saving data to files is as easy as reading it. We use the family of `to_*` methods to save data to different formats.

Example

```
df = pd.DataFrame({'a': [1, 2], 'b': [3, 4]})
df.to_csv('data.csv')
df.to_excel('data.xlsx')
df.to_json('data.json')w
```

There are plenty of options to customize the output, like the separator used in CSV files, the compression level, or the format of the JSON file, depending on the format you are saving to. Check the documentation for more information.

3.3 Basic data exploration

At this point, we have the ability to load into a Python script a dataset as large as our computer memory can handle, which can be comprised of many millions of rows and columns. Let us learn a couple of ways to get basic information about the data we have loaded.

We will use this dataset with XKCD comic information for a test. It has around 3k lines.

```
import pandas as pd

df = pd.read_json("hf://datasets/olivierdehaene/xkcd/dataset.jsonl", lines=True)
```

To start, we can try to just print the dataset:

```
print(df)
```

Output

```
        id  ...                                        explanation
0        1  ...  The comic shows a young boy floating in a barr...
1        2  ...  This comic does not present a particular point...
2        3  ...  This comic does not present a particular point...
3        4  ...  This comic does not present a particular point...
4        5  ...  This comic is a mathematical and technical jok...
...    ...  ...                                                ...
2625  2626  ...  In binary computing, 16 bit unsigned numbers r...
2626  2627  ...  Electron microscopes , electron telescopes and...
2627  2628  ...  This is analogous to something much more commo...
2628  2629  ...  The Willis Tower (formerly the Sears Tower) is...
2629  2630  ...  The Space Shuttle was a reusable spacecraft sy...

[2630 rows x 8 columns]
```

3.3.1 Head and tail

The dataset is so large that Pandas has to omit most of the rows and columns, showing "..." instead. We can force pandas to print it all by using the `to_string` method, but this would just blast our screen with information. Instead, we can use the `head` and `tail` methods to show the first and last rows of the dataset, respectively.

```
pd.set_option('display.max_columns', 3)
print("First two rows:")
print(df.head(2))
print("Last 2 rows:")
print(df.tail(2))
```

Output

```
First two rows:
   id  ...                                        explanation
0   1  ...  The comic shows a young boy floating in a barr...
1   2  ...  This comic does not present a particular point...

[2 rows x 8 columns]
Last 2 rows:
        id  ...                                        explanation
2628  2629  ...  The Willis Tower (formerly the Sears Tower) is...
2629  2630  ...  The Space Shuttle was a reusable spacecraft sy...

[2 rows x 8 columns]
```

Head and tail gives us a quick glimpse of the first and last rows, but in this case the output is still not very useful, as there are too many columns with too much text to fit. We still do not know what kind of information is stored for each entry.

3.3.2 Info

The `info` method gives us a concise summary of the DataFrame. It shows the number of non-null entries in each column, the data type of each column, and the memory usage of the DataFrame.

```
print(df.info())
```

Output

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2630 entries, 0 to 2629
Data columns (total 8 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   id              2630 non-null   int64
 1   title           2630 non-null   object
 2   image_title     2622 non-null   object
 3   url             2630 non-null   object
 4   image_url       2628 non-null   object
 5   explained_url   2630 non-null   object
 6   transcript      2628 non-null   object
 7   explanation     2625 non-null   object
dtypes: int64(1), object(7)
memory usage: 164.5+ KB
None
```

Now we know that the dataset has 2630 entries, and that most columns have no missing values. We also know the name and data type of each column.

3.4 Basic DataFrame operations

Just for fun, lets download the image of a random comic in the dataset. We will use the `requests` library to download the image, which will be rendered below the code.

```
import requests
import numpy as np
id = np.random.randint(0, len(df))
response = requests.get(df['image_url'].iloc[id])
with open('xkcd.png', 'wb') as f:
      f.write(response.content)
```

There are two new DataFrame operations in this example. For one, we used the `len` function to get the number of rows in the DataFrame. Being a kind of container, a DataFrame has compatibility with the usual Python functions and operators that work with containers (like `list`).

Secondly we used the `[]` operator to access a column of the DataFrame. This returns a Series object, which we can index with an integer to get the value of a specific row.

Info

There are 3 options to access the data of a pandas object:

- `[]`, the "standard getter" which is highly overloaded
- `.loc[]`, label based selector
- `.iloc[]`, integer location based selector

We will explore these in more detail in next lessons.

Advice

The `[]` operator is highly overloaded and its use is discouraged. Use `.loc[]` and `.iloc[]` whenever possible.

3.4.1 Names and labels

In the XKCD dataset, the columns have names like `image_url` and `transcript`. These names are used to identify the columns in the DataFrame. We can access the names of the columns with the `.columns` attribute:

```
print(df.columns)
```

Output

```
Index(['id', 'title', 'image_title', 'url', 'image_url', 'explained_url',
       'transcript', 'explanation'],
      dtype='object')
```

And we can access the index of the DataFrame with the `.index` attribute:

```
print(df.index)
```

Output

```
RangeIndex(start=0, stop=2630, step=1)
```

Which in this case are just the integers from 0 to 2629. Furthermore, the dataset contains a column named `id` which is a unique identifier for each comic, making the index redundant. We can set this column as the index of the DataFrame with the `.set_index` method:

```
df.set_index('id', inplace=True)
print(df.info())
```

Output

```
<class 'pandas.core.frame.DataFrame'>
Index: 2630 entries, 1 to 2630
Data columns (total 7 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   title          2630 non-null   object
 1   image_title    2622 non-null   object
 2   url            2630 non-null   object
 3   image_url      2628 non-null   object
 4   explained_url  2630 non-null   object
 5   transcript     2628 non-null   object
 6   explanation    2625 non-null   object
dtypes: object(7)
memory usage: 164.4+ KB
None
```

Now the DataFrame has the `id` column as the index, and the `RangeIndex` has disappeared. The `read_json` method also has an `index_col` argument that can be used to set the index column when reading the data.

DataFrame allows to set names for the columns and the index, and allows to give a name to the columns as a whole.

```
#Rename the columns
df.columns = ['Title', 'Image Title', 'URL', 'Image URL', 'Explained URL',
              'Transcript', 'Explanation']
# Rename the index "column"
df.index.name = 'ID'
df.columns.name = 'Properties'
print(df.head(2))
```

Output

```
Properties                 Title  ...  \
ID                                ...
1                Barrel - Part 1  ...
2           Petit Trees (sketch)  ...

Properties                                          Explanation
ID
1           The comic shows a young boy floating in a barr...
2           This comic does not present a particular point...

[2 rows x 7 columns]
```

We can also rename columns using the `.rename` method.

```
print(df.columns)
df.rename(columns={'Title': 'Comic Title'}, inplace=True)
print(df.columns)
```

Output

```
Index(['Title', 'Image Title', 'URL', 'Image URL', 'Explained URL',
       'Transcript', 'Explanation'],
      dtype='object', name='Properties')
Index(['Comic Title', 'Image Title', 'URL', 'Image URL', 'Explained URL',
       'Transcript', 'Explanation'],
      dtype='object', name='Properties')
```

3.4.2 Dropping data

Often, we will want to remove columns or rows from a DataFrame. We can do this with the `.drop` method. For instance, to remove the `URL` column:

```
df.drop(columns=['URL'], inplace=True)
# Without inplace=True, the method returns a new DataFrame
# We would have to write:
# df = df.drop(columns=['URL'])
print(df.columns)
```

Output

```
Index(['Comic Title', 'Image Title', 'Image URL', 'Explained URL',
       'Transcript', 'Explanation'],
      dtype='object', name='Properties')
```

We can also remove rows by passing the index of the rows to remove:

```
df = df.drop(index=[1, 2])
print(df.head(2))
```

Output

```
Properties         Comic Title  ...  \
ID                                 ...
3                 Island (sketch)  ...
4               Landscape (sketch)  ...

Properties                                              Explanation
ID
3           This comic does not present a particular point...
4           This comic does not present a particular point...

[2 rows x 6 columns]
```

4 Exercises

4.1 Data migrator

Goal

We will write a Python script that will migrate data from a CSV file to a series of formats. The script will use the name of the input file and the name of the output file. The script will read the data from the input file and save it to the output file in the following formats:

- A CSV file
- An Excel file
- A JSON file

At the end, your script should be able to be called like this:

```
$ python data_migrator.py
The data has been migrated to from data.csv to data.xlsx
```

Milestone

Open the terminal, activate the conda environment, and navigate to the folder where you want to save the script. Use the `code` command to create a new Python script named `data_generator.py`.

Milestone

Make `data_generator.py` create an initial dataset to test the final script. The dataset should be a CSV file with at least some rows and columns. Save the dataset as `data.csv`. Name the columns, index and the DataFrame itself. Call the script using the terminal when you are done to check that the dataset is created correctly.

hint

Use any of the `DataFrame` creation methods we have seen so far, for instance, a 2D numpy array or a dictionary of lists.

Milestone

Write the `data_migrator.py` script (call `code` again from the terminal). Make it so that the script reads "data.csv" and saves it to "data.xlsx". Call the script using the terminal to check that the data is saved correctly.

Advanced milestone

Use `argparse` to add the `--format` argument to the script. The argument should accept the values "csv", "xlsx", "json". Your script should be callable as:

```
$ python data_migrator.py --input "data.csv" --format "xlsx"
The data has been migrated to data.xlsx
```

4.2 Population finder

Goal

We will write a Python script that will tell us the current population of a given city. If the city is not found, the script will return an error message.
We will use the `worldcities.csv` file, which contains the population of many cities around the world. The file is available here.
At the end, your script should be able to be called like this:

```
$ python population_finder.py
The population of New York is 8,175,133
```

Milestone

Open the terminal, activate the conda environment, and navigate to the folder where you want to save the script. Use the `code` command to create a new Python script named `population_finder.py`.

Milestone

Make `population_finder.py` read the `worldcities.csv` file and print the first and last few rows to check that the data is loaded correctly. Make the script also print some basic information about the DataFrame.

Advanced milestone

Using `argparse`, add the `--city` argument to the script. The argument should accept the name of a city. Make the program print the population of the city. The program should exit with an error message if the city is not found.

Advanced milestone

Add a `--country` argument to the script. If the `--country` argument is provided, the program should print the population of the largest city in the country.

```
$ python population_finder.py --country "Japan"
The population of Tokyo is 13,833,000
```

The program should exit with an error message if the country is not found.