Pandas II & III

Raul P. Pelaez

February 3, 2026

Contents

1 Introduction

Let us continue exploring the Pandas library. Thus far, we have learned about the Pandas `Series` and `DataFrame` data structures. We have seen how to load and get basic information about a dataset. In this lesson, we will explore indexing and selection more in depth, as well as data manipulation techniques. We will also cover advanced operations such as aggregation, grouping, and merging.

2 Indexing and Selection

Last session we covered both `.loc` (label-based indexing) and `.iloc` (integer-based indexing). Today we will explore more advanced indexing techniques, such as boolean indexing and slicing. The behavior of these methods is similar to NumPy arrays, but with the added functionality of Pandas DataFrames.

Info

Selecting a column

To select a column from a DataFrame, you can use the following syntax:

```
import pandas as pd
data = {'name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
        'age': [25, 30, 35, 40, 45],
        'city': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Phoenix']}
df = pd.DataFrame(data)
print(df.loc[:, 'name'])
```

Output

```
0      Alice
1        Bob
2    Charlie
3      David
4        Eve
Name: name, dtype: object
```

Advice

We could also use just the `[]` operator to select a column. For example, `df['name']`. However, the `[]` operator can sometimes behave in an ambiguous way, so it is recommended to use the `.loc` method for clarity.

The first argument to `loc`, `:`, specifies that we want to select all rows. The second argument, `name`, specifies the column we want to select. The output will be a Pandas `Series` containing the values of the `name` column.

Info

Selecting multiple columns

The `.loc` method can also be used to select multiple columns. To do this, pass a list of column names as the second argument to `loc`. For example:

```
print(df.loc[2:4, ['name', 'age']])
```

Output

```
      name  age
2  Charlie   35
3    David   40
4      Eve   45
```

Here, we are selecting rows 2 through 4 and columns `name` and `age`. The output will be a Pandas `DataFrame` containing the specified rows and columns.

Info

Slicing

Slicing is the process of selecting a subset of data from an array-like entity, such as a `Series` or a `DataFrame`. When we write something like:

```
df.loc[2:4, 'name']
```

We are "slicing" the `DataFrame`.

Advice

The index is "just" another column

We can select elements in the index like any other column. For example:

```
import pandas as pd
df = pd.DataFrame({"legs": [2, 4, 100],
                   "wings": [2, 0, 0]},
                  index=["colibri", "centipide", "dog"])
print(df.loc[["colibri", "centipide"], ["legs", "wings"]])
```

Output

```
           legs  wings
colibri       2      2
centipide     4      0
```

You can think about `loc` as a method that receives "row" and "column" arguments. Each of these arguments can be a single value, a list of values, or a slice.

The `loc` function will assume, if it only receives one argument that is not a list, that it is a row argument. For example:

```
print(df.loc["colibri"])
```

Output

```
legs     2
wings    2
Name: colibri, dtype: int64
```

Info

Accessing by integer position

The `.iloc` method is used to access elements by integer position. For example:

```
import pandas as pd
df = pd.DataFrame({"A": [1, 2, 3, 4, 5],
                   "B": [10, 20, 30, 40, 50]},
                  index=["a", "b", "c", "d", "e"])
print(df.iloc[2:4, 0:2])
```

Output

```
   A   B
c  3  30
d  4  40
```

Info

Boolean indexing

Boolean indexing allows you to select data based on a condition. For example:

```
import pandas as pd
df = pd.DataFrame({"A": [1, 2, 3, 4, 5],
                   "B": [10, 20, 30, 40, 50]},
                  index=["a", "b", "c", "d", "e"])
print(df[df.loc[:,"A"] > 3])
```

Output

```
   A   B
d  4  40
e  5  50
```

This will return all rows where the value in column `A` is greater than 2.
Think about the expression `df["A"] > 3` as an array of booleans, where each element is `True` if the corresponding element in the column `A` is greater than 3, and `False` otherwise. Similarly to NumPy, we can use this array to index the DataFrame.

Warning

Chaining conditions

When chaining conditions, use the bitwise operators `&` (and) and `|` (or) instead of the logical operators `and` and `or`. For example:

```
print(df[(df.loc[:,"A"] > 1) & (df.loc[:,"A"] < 4)])
```

Output

```
   A   B
b  2  20
c  3  30
```

If we were to write `df["A"] > 1 and df["A"] < 4`, we would be trying to apply the logical operator `and` to two arrays (like saying "a bunch of elements AND a bunch of elements" which makes no sense), instead of applying the and operation element-wise (which can be read as "a condition AND a condition, for all conditions in the lists").

Advice

Try to be explicit

When selecting values from a `DataFrame`, try to be as explicit as possible in the code about your intention. For example, instead of writing:

```
df["A"]
# Or equivalently
df.A
```

Write:

```
df.loc[:, "A"]
```

The workings of the `[]` operator can be ambiguous and unexpected, and it is better to use the `loc` method for clarity.

Info

Setting values

Whenever we select a subset of a `DataFrame` (including a single element), we can set or modify the values of that subset. For example:

```
import pandas as pd
import numpy as np
df = pd.DataFrame({"A": list(range(5)),
                   "B": np.array(range(5))*2+1},
                  index=list("abcde"))
df.loc["a", "A"] = 1001
print(df)
```

Output

```
      A  B
a  1001  1
b     1  3
c     2  5
d     3  7
e     4  9
```

3 Data Manipulation

Often, we will need to apply some function to all elements of a column (or a row, or the entire `DataFrame`). Naively, we might try to iterate over the elements in order to do so:

```
import pandas as pd
df = pd.DataFrame({"A": [1, 2, 3],
                   "B": [10, 20, 30]},
                  index=["a", "b", "c"])
# This is a very inefficient way to multiply all elements by 200
column = df.loc[:, "A"]
for i in column.index:
    column[i] = column[i] * 200
print(df)
```

Output

```
     A   B
a  200  10
b  400  20
c  600  30
```

Advice

Python is slow
Python, being an interpreted language, is inherently slow and inefficient. Thus, we try to avoid writing our logic in pure Python whenever possible (for instance, avoiding loops). Instead, we try to think "algorithmically" or "vectorially" (i.e., in terms of operations on entire arrays), transforming our logic into NumPy or Pandas operations.
Doing this will make our code faster and more readable, and perhaps more importantly, "Pythonic", which is a term that refers to writing code that is idiomatic to the language. Critically, this makes our code easier to understand (i.e. *expected*) for other Python developers.

On the other hand, expressing our logic in terms of pandas/numpy operations tends to be more efficient (we are talking orders of magnitude) because these libraries are written in C, and they are optimized for these kinds of operations. This means that our scripts, instead of being "a sequence of Python instructions", are more like "drivers" for the underlying C code, which is how one should write Python.

Write the code for which performance is irrelevant in pure Python, and use library calls for the performance-critical parts. For instance, if you need to process a list of datasets lying in your hard drive, you would use a Python for loop to list and iterate over the files, but you would use Pandas to load and process each dataset. This way you get the best of both worlds: the flexibility and ease of use of Python, and the performance of C.

Instead, we should use the `apply` method to apply a function to all elements of a column.

Info

Applying functions element-wise
The `apply` method allows us to apply a function to each element of a column. For example:

```
import pandas as pd
df = pd.DataFrame({"A": [1, 2, 3],
                   "B": [10, 20, 30]},
                  index=["a", "b", "c"])
# Apply does not modify the original DataFrame, rather produces a copy
df.loc[:,"A"] = df.loc[:,"A"].apply(lambda x: x * 200)
print(df)
```

Output

```
     A   B
a  200  10
b  400  20
c  600  30
```

Note that we did not changed the type of the column, we just multiplied all elements by 200. Try to multiply by 200.5 instead, you will get a warning. In trying to be explicit, if our intention is to modify the type of the column, we should use the `astype` method instead (as we explore in the next section).

Info

Lambda functions
Lambda functions are small, unnamed (anonymous) functions that can have any number of arguments but only one expression. The result of this expression is the return value of the function. For example:

```
f = lambda x: x * 2
print(f(3)) # Output: 6
```

Lambda functions are useful when you need a simple function for a short period of time. They are often used as arguments to higher-order functions (functions that take other functions as arguments), such as `apply`.

Info

Creating new columns

We can create new columns in a `DataFrame` by assigning a value to a new column name. For example:

```
import pandas as pd
df = pd.DataFrame({"A": [1, 2, 3],
                   "B": [10, 20, 30]},
                  index=["a", "b", "c"])
df.loc[:,"C"] = df.loc[:,"A"]  + df.loc[:,"B"]
print(df)
```

Output

```
   A   B   C
a  1  10  11
b  2  20  22
c  3  30  33
```

This will create a new column `C` in the `DataFrame` that is the sum of columns `A` and `B`.

3.1 Data type conversions

Converting data types is a common operation when working with data. For example, we might need to convert a column of strings to integers or floats. Or strings to dates. Or "objects" to strings or categorical data.

Info

Converting data types

The `astype` method allows us to convert the data type of a column. For example:

```
import pandas as pd
df = pd.DataFrame({"A": ["1", "2", "3"],
                   "B": [10, 20, 30]},
                  index=["a", "b", "c"])
df.loc[:,"A"] = df.loc[:,"A"].astype(int)
print(df)
```

Output

```
   A   B
a  1  10
b  2  20
c  3  30
```

Warning

String is a special case

Converting a column to a string is a special case because Pandas treats strings as objects. For example:

```
import pandas as pd
df = pd.DataFrame({"location": ["The Louvre", "MoMA"],
                   "artist": ["Leonardo da Vinci", "Vincent van Gogh"]},
                  index=["Mona Lisa", "The Starry Night"])
print(df.info())
```

Output

```
<class 'pandas.core.frame.DataFrame'>
Index: 2 entries, Mona Lisa to The Starry Night
Data columns (total 2 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   location  2 non-null      object
 1   artist    2 non-null      object
dtypes: object(2)
memory usage: 48.0+ bytes
None
```

The output of the `info` method shows that the `location` column is of type `object`. To treat them as a string, we can use the `str` accessor, which allows us to apply string methods to the elements of the column. For example:

```
# Split the artist strings into lists, using the space character as the separator
artists_str = df.loc[:,"artist"].str.split()
print(artists_str)
```

Output

```
Mona Lisa            [Leonardo, da, Vinci]
The Starry Night      [Vincent, van, Gogh]
Name: artist, dtype: object
```

If we really want the list of strings for that column, we can use the `list()` function:

```
artists_list = list(df.loc[:,"artist"])
print(artists_list)
```

Output

```
['Leonardo da Vinci', 'Vincent van Gogh']
```

3.1.1 Time series

Pandas was originally designed for financial data analysis, so it has excellent support for time series data. Pandas has an extensive and intricate system for understanding dates and times, and it is very flexible. For example:

```
import pandas as pd
import numpy as np
import datetime
unix_time_origin = 0 # 1-1-1970
dti = pd.to_datetime(
    ["1/3/2018",
     "17-09-1985",
     np.datetime64("2018-07-01"),
     datetime.datetime(2018, 6, 3),
     unix_time_origin],
    format="mixed"
)
```

```
print(dti)
print(f" Days:   {[t.day for t in dti]}")
print(f" Months: {[t.month for t in dti]}")
```

Output

```
DatetimeIndex(['2018-01-03', '1985-09-17', '2018-07-01', '2018-06-03',
               '1970-01-01'],
              dtype='datetime64[ns]', freq=None)
 Days:   [3, 17, 1, 3, 1]
 Months: [1, 9, 7, 6, 1]
```

In the spirits of Python, Pandas tries to be as flexible as possible. Anything that could intuitively be interpreted as a date Pandas will convert correctly. For example, the second entry in the list is a date in the format "dd-mm-yyyy", which is not the default format for Pandas. However, Pandas is smart enough to understand that 17 cannot be a month, so it switches the day and month automatically.

The `format=mixed` argument tells Pandas to try to interpret the date in the most flexible way possible on a per-element basis. This is useful when you have a mix of date formats in your data (which happens). If the format is consistent, but somehow ambiguous, the `format` argument can be used to specify the format explicitly, check the documentation for `to_datetime` for more information. For instance, if we know that the date is in a very specific format (which might include the time of day too), we can specify it:

```
import pandas as pd
dti = pd.to_datetime("1.3//2018 13.45:12", format="%d.%m//%Y %H.%M:%S")
print(f" Day:  {dti.day}, Month: {dti.month}, Year: {dti.year}")
print(f"Hour: {dti.hour}, Minute: {dti.minute}, Second: {dti.second}")
```

Output

```
 Day:  1, Month: 3, Year: 2018
Hour: 13, Minute: 45, Second: 12
```

Advice

Usually, we read a dataset from a file, and the dates are read as strings. We can convert them to dates using the `to_datetime` function. For example:

```
import pandas as pd
df = pd.DataFrame({"date": ["2021-01-01", "2021-01-02", "2021-01-03"],
                   "value": [10, 20, 30]})
# This construct is common
df.loc[:,"date"] = pd.to_datetime(df.loc[:,"date"])
print(df)
```

Output

```
                 date  value
0  2021-01-01 00:00:00     10
1  2021-01-02 00:00:00     20
2  2021-01-03 00:00:00     30
```

4 Advanced Operations

Advanced

Pivoting with `melt`

The `melt` method is used to transform a wide DataFrame into a long DataFrame. For example:

```
import pandas as pd

df = pd.DataFrame(
    {
        "first": ["John", "Mary"],
        "last": ["Doe", "Bo"],
        "job": ["Nurse", "Economist"],
        "height": [5.5, 6.0],
        "weight": [130, 150],
    }
)
melt_df = df.melt(
    id_vars=["first", "last"],
    var_name="quantity",
    value_vars=["height", "weight"]
)

print("\n Unmelted: ")
print(df)
print("\n Melted: ")
print(melt_df)
```

Output

```
 Unmelted:
  first last        job  height  weight
0  John  Doe      Nurse     5.5     130
1  Mary   Bo  Economist     6.0     150

 Melted:
  first last quantity  value
0  John  Doe   height    5.5
1  Mary   Bo   height    6.0
2  John  Doe   weight  130.0
3  Mary   Bo   weight  150.0
```

We "melted" the columns `height` and `weight` into a single column `quantity`. Try adding "job" to the `id_vars` or the `value_vars` and see what happens.

4.1 Aggregation

Aggregation refers to operations that involve all values in a column (or more than one column) to produce a single value. For example, calculating the mean, sum, or count of a column.

Info

Basic statistics

Aggregation involves calculating summary statistics (e.g., mean, sum, count) for a dataset. For instance:

```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.randint(0, 11, (1000, 2)),
                  columns=["value1", "value2"])
print(df.agg({"value1": ["mean", "sum", "count"]}))
```

Output

```
         value1
mean      5.094
sum    5094.000
count  1000.000
```

We can also use the individual aggregation functions directly:

```
print(df["value2"].mean())
print(df["value2"].sum())
print(df["value2"].count())
```

Output

```
4.935
4935
1000
```

Info

Describing data

The `describe` method provides a summary of the numerical data:

```
print(df["value2"].describe())
```

Output

```
count    1000.000000
mean        4.935000
std         3.233145
min         0.000000
25%         2.000000
50%         5.000000
75%         8.000000
max        10.000000
Name: value2, dtype: float64
```

Info

Counting values
The `value_counts` method is used to count the number of occurrences of each unique value in a column. For example:

```
print(df["value1"].value_counts())
```

Output

```
value1
6     112
10    110
3      97
0      92
8      89
1      87
4      86
9      83
7      82
5      81
2      81
Name: count, dtype: int64
```

Essentially, this produces an histogram of the values in the column.

4.2 Grouping and Merging

Info

Grouping data: `groupby`
Grouping allows for performing calculations on subsets of data with some common characteristic. For example:

```
import pandas as pd
data = {
    "name": ["Alice", "Bob", "Charlie", "David", "Eve"],
    "city": ["New York", "New York", "Chicago", "New York", "Chicago"],
    "income": [50000, 60000, 70000, 80000, 90000],
}
df = pd.DataFrame(data)
grouped = df.groupby("city")
print(grouped["income"].mean())
```

Output

```
city
Chicago     80000.000000
New York    63333.333333
Name: income, dtype: float64
```

In this example, we group the data by the `city` column and calculate the mean income for each city. In a single line, we did something equivalent the following:

- Generate datasets with all the rows that have the same value in the `city` column.
- For each dataset, calculate the mean of the `income` column.
- Create a `Series` with the means, indexed by the unique values in the `city` column.

Info

Merging DataFrames: `merge`

When merging two tables, we stitch them together on the basis of two columns (one from the first table, another from the second table) having the same value. The resulting merged table will be the first table glued with the second table, with the shared column acting as glue. There are four types of joins:

- Inner join (the default): "And" operation, return rows present in both tables.
- Left join: Return all rows in left table, rows in right table without a match will have NaN.
- Right join: Return all rows in right table, rows in left table without a match will have NaN.
- Outer join: All the rows for both tables. NaN when there is no match.

For example:

```
import pandas as pd
df_left = pd.DataFrame({"value": [1, 2], "name": ["A", "B"]})
df_right = pd.DataFrame({"value": [22, 33], "name": ["B", "C"]})
print(f"df_left:\n{df_left}")
print(f"df_right:\n{df_right}")
print(f"Inner join:\n{pd.merge(df_left, df_right, on="name", how="inner")}")
print(f"Left join:\n{pd.merge(df_left, df_right, on="name", how="left")}")
print(f"Right join:\n{pd.merge(df_left, df_right, on="name", how="right")}")
print(f"Outer join:\n{pd.merge(df_left, df_right, on="name", how="outer")}")
```

Output

```
df_left:
   value name
0      1    A
1      2    B
df_right:
   value name
0     22    B
1     33    C
Inner join:
   value_x name  value_y
0        2    B       22
Left join:
   value_x name  value_y
0        1    A      NaN
1        2    B     22.0
Right join:
   value_x name  value_y
0      2.0    B       22
1      NaN    C       33
Outer join:
   value_x name  value_y
0      1.0    A      NaN
1      2.0    B     22.0
2      NaN    C     33.0
```

5 Exercises

5.1 Meteo

Goal

Public (as in coming from some official state organism) APIs often include weather data among the available queries. Unfortunately, the data is often not in the best shape, moreover being presented in far from standard formats.
For this example, I scraped the daily measures of a weather station in a city called "Colmenar viejo" for every day since some point around the 80s (which I reckon is the birth of the station) until today. The data is in a CSV file, but it has a series of issues that you will need to fix before being able to correctly analyze it.
Our goal for today is to load the data, clean it, and analyze it. The data is in the file "meteo.csv", download it from BlackBoard. As usual, we will get there milestone by milestone.

Learning goal

- This exercise is designed to train you in a real life scenario where you are presented with a dirty dataset.
- You will need to do some exploratory work to understand the data, and decide on which is the best way to clean it.
- There are a million ways to go about solving each problem, but know that, explicitly or not, the notes thus far have given you the tools to solve this exercise.
- This exercise, by being "vaguer" than the previous ones, is also trying to encourage you to learn how to ask the right questions, and how to read the documentation to find the answers.
- The dataset was scraped almost as-is. The API responded with the fields that you can find as columns in the CSV file. No additional context or explanation was provided.

Milestone

1. Start a new terminal, navigate to a new directory and open a script called "meteo.py".
2. Download the meteo data and place it in the same directory as the script.
3. For the exploratory analysis, a Jupyter notebook will come in handy, so start one in the same directory.
4. Whenever you learn something definitive about the data, write it down in the script.

Milestone

Load the data into a DataFrame and use the `info`, `head` and `describe` methods (as well as any other exploratory method you remember or discover) to get a sense of the data. In particular, look for:

- The column that contains the date information
- The column that contains the average temperature

Milestone

- Create a new empty `DataFrame`, in which you will store the cleaned data.
- Transform the column containing the date information into a `datetime` object.
- Add the column containing the date information to the new `DataFrame`.

Milestone

- Transform the column containing the average temperature into the `float` type.
- Add the column containing the average temperature to the new `DataFrame`.

hint

- The column was read as a string because of the presence of commas in the numbers.
- Does every row have an actual value? If not, what should you do with the missing values?
- Look at the documentation for `read_csv` to see if there is an argument that can help you with this.

Advanced milestone

There are several ways to transform the column containing the average temperature into a `float` type. The most direct is perhaps through `read_csv`. Instead, implement the functionality in two different ways:

- Using the `apply` method.
- Using the `.str` accessor, its replace method, and `astype`.

Milestone

Plot the average temperature over time using the following function:

```
def plot_temperature(date: pd.Series, temperatures: pd.Series):
    assert len(date) == len(temperatures)
    assert date.dtype == np.dtype("datetime64[ns]")
    assert temperatures.dtype == np.dtype("float64")
    import matplotlib.pyplot as plt
    years = date.dt.year
    day = date.dt.dayofyear
    plt.scatter(day, temperatures, c=years, marker=".")
    plt.colorbar(label="Year")
    plt.xticks(rotation=45)
    plt.xlabel("Date")
    plt.ylabel("Temperature (Celsius)")
    plt.show()
```

Milestone

Group the data by year and calculate the maximum and minimum temperature for each year. Create a new DataFrame with this data. Plot or print the results. Using that dataset, find and print the years with the highest and lowest maximum temperatures.

hint

- It might be a good idea to add a new column to the DataFrame with the year of each date.
- Use the `groupby` method to group the data by year.

Milestone

Add an option to the script that allows to get the average temperature for a given date. The script should print the average temperature for the given date, your script should work like this:

```
$ python meteo.py --date 2021-01-01
The average temperature for 2021-01-01 was 10.5 degrees.
```

Advanced milestone

Can you find the time of day for which the minimum temperature was recorded each day? Store in two arrays the date and time of day for which the minimum temperature was recorded. Send them both through `pd.to_datetime`. Ask ChatGPT to produce a function that receives both arrays and plots the time of day for which the minimum temperature was recorded each day.

hint

- Use the `dayofyear` attribute of the datetime object to get the day of the year, so you can plot the time of day for which the minimum temperature was recorded each day for all years in the same plot.