Scientific Computing with Python: Pandas

Pandas

Objectives

- To be able to use the Python library Pandas for scientific computing.
- To understand the data type of DataFrame.
- To be able to use the Python library Pandas to read and write data.
- To be able to write programs that uses Pandas for analyzing data.

What is Pandas?

- Pandas is a fast, powerful, flexible and easy to use open source Python package that is most widely used for data science/data analysis and machine learning tasks.
- Pandas is built on top of Numpy, which provides support for multi-dimensional arrays.
  - A lot of the structure of NumPy is used or replicated in Pandas.
  - Data in pandas is often used to feed statistical analysis in SciPy, plotting functions from Matplotlib, and machine learning algorithms in Scikit-learn.

Why Use Pandas?

- Pandas allows us to analyze big data and make conclusions based on statistical theories.
- Pandas can clean messy data sets, and make them readable and relevant.
- Relevant data is very important in data science.

What Can Pandas Do?

- Pandas makes it simple to do many of the time consuming, repetitive tasks associated with working with data, including:
  - Data cleaning
  - Data fill
  - Data normalization
  - Merges and joins
  - Data visualization
  - Statistical analysis
  - Data inspection
  - Loading and saving data
  - And much more
- In fact, with Pandas, you can do everything that makes world-leading data scientists vote Pandas as the best data analysis and manipulation tool available.

Installation of Pandas

- To check if you already have Pandas installed in your Python installation, run the command:

```
import pandas as pd
```

- If no error message is returned, that's a good sign NumPy is already available.
- Pandas needs to be installed first if you get an error message like

```
ModuleNotFoundError: No module named 'pandas'
```

Installation of Pandas

- Open a common window
- If you use the pip Python package manager, the required command is

```
pip install pandas
```

- If you use Anaconda to mange packages, the required command is

```
conda install pandas
```

- If you need more detailed installation instructions, refer to https://pandas.pydata.org/.

Series and DataFrames

- The primary two components of pandas are the **Series** and **DataFrame**
- A **Series** is essentially a column
- A **DataFrame** is a multi-dimensional table made up of a collection of Series.

Series

	apples
0	3
1	2
2	0
3	1

+

Series

	oranges
0	0
1	3
2	7
3	2

=

DataFrame

	apples	oranges
0	3	0
1	2	3
2	0	7
3	1	2

Creating DataFrames

- Let's say we have a fruit stand that sells apples and oranges. We want to have a column for each fruit and a row for each customer purchase.
- To organize this as a dictionary for pandas we could first build a dictionary, and then pass it to the pandas DataFrame constructor

```
data = {'apples': [3, 2, 0, 1],
        'oranges': [0, 3, 7, 2]}

purchases = pd.DataFrame(data)
```

OUT:

	apples	oranges
0	3	0
1	2	3
2	0	7
3	1	2

Creating DataFrames

- How did that work?
  - Each (key, value) item in data corresponds to a column in the resulting DataFrame.
  - The Index of this DataFrame was given to us on creation as the numbers 0-3, but we could also create our own when we initialize the DataFrame.

```
data = {'apples': [3, 2, 0, 1],
            'oranges': [0, 3, 7, 2]}

purchases = pd.DataFrame(data, index=['June',
  'Robert', 'Lily', 'David'])
```

OUT:

	apples	oranges
June	3	0
Robert	2	3
Lily	0	7
David	1	2

DataFrames Indexing/Selection

- We could locate a customer's order by using their name

```
>>>purchases.loc['June']
apples      3
oranges     0
Name: June, dtype: int64

>>>purchases.loc['June', 'apples’]
3
```

OUT:

	apples	oranges
June	3	0
Robert	2	3
Lily	0	7
David	1	2

DataFrames Indexing/Selection

- The basics of indexing are as follows

Operation	Syntax	Result
Select column	`df[col]`	Series
Select row by label	`df.loc[label]`	Series
Select row by integer location	`df.iloc[loc]`	Series
Slice rows	`df[5:10]`	DataFrame
Select rows by boolean vector	`df[bool_vec]`	DataFrame

DataFrames Indexing/Selection

- Pandas objects have a number of attributes enabling you to access the metadata
  - Shape: gives the axis dimensions of the object, consistent with ndarray
  - Axis labels
    - Series: index (only axis)
    - DataFrame: index (rows) and columns
- Note, these attributes can be safely assigned to!

```
In [5]: import pandas as pd
        import numpy as np

In [7]: index = pd.date_range("1/1/2000", periods=8)
        df = pd.DataFrame(np.random.randn(8, 3), index=index, columns=["A", "B", "C"])

In [8]: df
Out[8]:
```

	A	B	C
2000-01-01	-1.935869	0.071922	-0.858021
2000-01-02	-0.734141	-0.287258	0.559143
2000-01-03	1.144963	-0.088131	-0.394950
2000-01-04	-0.816180	-0.716559	-2.541444
2000-01-05	-1.501135	0.438521	0.123611
2000-01-06	0.824147	0.711382	0.112720
2000-01-07	-1.465689	-0.057186	-0.207795
2000-01-08	0.641109	-2.135995	0.290558

```
In [9]: df[:2]
Out[9]:
```

	A	B	C
2000-01-01	-1.935869	0.071922	-0.858021
2000-01-02	-0.734141	-0.287258	0.559143

DataFrames Indexing/Selection

In [5]:
```
import pandas as pd
import numpy as np
```

In [7]:
```
index = pd.date_range("1/1/2000", periods=8)
df = pd.DataFrame(np.random.randn(8, 3), index=index, columns=["A", "B", "C"])
```

In [8]:
```
df
```

Out[8]:

	A	B	C
2000-01-01	-1.935869	0.071922	-0.858021
2000-01-02	-0.734141	-0.287258	0.559143
2000-01-03	1.144963	-0.088131	-0.394950
2000-01-04	-0.816180	-0.716559	-2.541444
2000-01-05	-1.501135	0.438521	0.123611
2000-01-06	0.824147	0.711382	0.112720
2000-01-07	-1.465689	-0.057186	-0.207795
2000-01-08	0.641109	-2.135995	0.290558

In [10]:
```
df.columns = [x.lower() for x in df.columns]
```

In [11]:
```
df
```

Out[11]:

	a	b	c
2000-01-01	-1.935869	0.071922	-0.858021
2000-01-02	-0.734141	-0.287258	0.559143
2000-01-03	1.144963	-0.088131	-0.394950
2000-01-04	-0.816180	-0.716559	-2.541444
2000-01-05	-1.501135	0.438521	0.123611
2000-01-06	0.824147	0.711382	0.112720
2000-01-07	-1.465689	-0.057186	-0.207795
2000-01-08	0.641109	-2.135995	0.290558

Get Column Name

In [27]:
```
df
```

Out[27]:

	a	b	c
2000-01-01	0.387906	0.165505	-0.296642
2000-01-02	0.675343	-0.012634	-0.946212
2000-01-03	-0.510775	-0.301052	-0.072443
2000-01-04	1.918426	0.679467	-0.840432
2000-01-05	-0.888401	-1.960367	0.568562
2000-01-06	0.266800	-0.738583	-0.915223
2000-01-07	-0.079060	1.429034	-0.038778
2000-01-08	0.387059	0.366470	0.222605

In [24]:
```
df.columns
```

Out[24]:
```
Index(['a', 'b', 'c'], dtype='object')
```

In [26]:
```
df.columns.values
```

Out[26]:
```
array(['a', 'b', 'c'], dtype=object)
```

Reading Data from CSVs

- With CSV files all you need is a single line to load in the data

```
import pandas as pd
df = pd.read_csv('purchases.csv')
```

OUT:

	Unnamed: 0	apples	oranges
0	June	3	0
1	Robert	2	3
2	Lily	0	7
3	David	1	2

- CSVs don't have indexes like our DataFrames, so all we need to do is just designate the index_col when reading:

```
import pandas as pd
df = pd.read_csv('purchases.csv', index_col=0)
```

OUT:

	apples	oranges
June	3	0
Robert	2	3
Lily	0	7
David	1	2

Converting Back to a CSV

- Similar to the ways we read in data, pandas provides intuitive commands to save it

```
df.to_csv('new_purchases.csv')
```

- Similar to handle CSV, pandas support read and write txt, json, and database file.

From DataFrames to NumPy

- Use to_numpy() or numpy.asarray() to convert DataFrame to a NumYy array.

```
In [11]: df
Out[11]:
```

	a	b	c
2000-01-01	-1.935869	0.071922	-0.858021
2000-01-02	-0.734141	-0.287258	0.559143
2000-01-03	1.144963	-0.088131	-0.394950
2000-01-04	-0.816180	-0.716559	-2.541444
2000-01-05	-1.501135	0.438521	0.123611
2000-01-06	0.824147	0.711382	0.112720
2000-01-07	-1.465689	-0.057186	-0.207795
2000-01-08	0.641109	-2.135995	0.290558

```
In [12]: df.to_numpy()
Out[12]: array([[-1.93586895,  0.07192163, -0.85802131],
                [-0.73414098, -0.28725828,  0.55914341],
                [ 1.14496323, -0.08813069, -0.39495005],
                [-0.81618038, -0.71655881, -2.54144387],
                [-1.50113472,  0.43852112,  0.1236115 ],
                [ 0.82414717,  0.71138192,  0.11272006],
                [-1.46568885, -0.05718584, -0.20779524],
                [ 0.64110863, -2.13599481,  0.29055796]])
```

```
In [14]: np.asarray(df)
Out[14]: array([[-1.93586895,  0.07192163, -0.85802131],
                [-0.73414098, -0.28725828,  0.55914341],
                [ 1.14496323, -0.08813069, -0.39495005],
                [-0.81618038, -0.71655881, -2.54144387],
                [-1.50113472,  0.43852112,  0.1236115 ],
                [ 0.82414717,  0.71138192,  0.11272006],
                [-1.46568885, -0.05718584, -0.20779524],
                [ 0.64110863, -2.13599481,  0.29055796]])
```

Filling Values

- In Series and DataFrame, the arithmetic functions have the option of inputting a fill_value, namely a value to substitute when at most one of the values at a location are missing.
- For example, when adding two DataFrame objects, you may wish to treat NaN as 0 unless both DataFrames are missing that value, in which case the result will be NaN

Filling Values

```
In [42]: df
Out[42]:
        one       two     three
a  1.394981  1.772517       NaN
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d       NaN  0.279344 -0.613172

In [43]: df2
Out[43]:
        one       two     three
a  1.394981  1.772517  1.000000
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d       NaN  0.279344 -0.613172
```

```
In [44]: df + df2
Out[44]:
        one       two     three
a  2.789963  3.545034       NaN
b  0.686107  3.824246 -0.100780
c  1.390491  2.956737  2.454870
d       NaN  0.558688 -1.226343

In [45]: df.add(df2, fill_value=0)
Out[45]:
        one       two     three
a  2.789963  3.545034  1.000000
b  0.686107  3.824246 -0.100780
c  1.390491  2.956737  2.454870
d       NaN  0.558688 -1.226343
```

Filling Values

- Fill NA/NaN values using the specified method.

In [15]:
```
df = pd.DataFrame([[np.nan, 2, np.nan, 0],
                   [3, 4, np.nan, 1],
                   [np.nan, np.nan, np.nan, np.nan],
                   [np.nan, 3, np.nan, 4]],
                  columns=list("ABCD"))
```

In [16]:
```
df
```

Out[16]:

	A	B	C	D
0	NaN	2.0	NaN	0.0
1	3.0	4.0	NaN	1.0
2	NaN	NaN	NaN	NaN
3	NaN	3.0	NaN	4.0

In [17]:
```
df.fillna(0)
```

Out[17]:

	A	B	C	D
0	0.0	2.0	0.0	0.0
1	3.0	4.0	0.0	1.0
2	0.0	0.0	0.0	0.0
3	0.0	3.0	0.0	4.0

In [18]:
```
values = {"A": 0, "B": 1, "C": 2, "D": 3}
df.fillna(value=values)
```

Out[18]:

	A	B	C	D
0	0.0	2.0	2.0	0.0
1	3.0	4.0	2.0	1.0
2	0.0	1.0	2.0	3.0
3	0.0	3.0	2.0	4.0

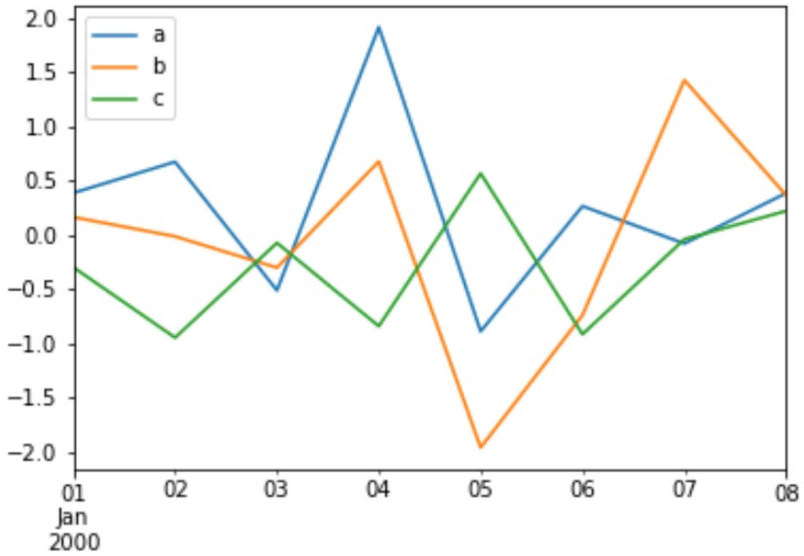
Plotting

- Pandas uses the plot() method to create diagrams.
- We can use Pyplot, a submodule of the Matplotlib library to visualize the diagram on the screen.
  - a scatter plot with the kind argument (kind = 'scatter’) and specifying the x and y arguments

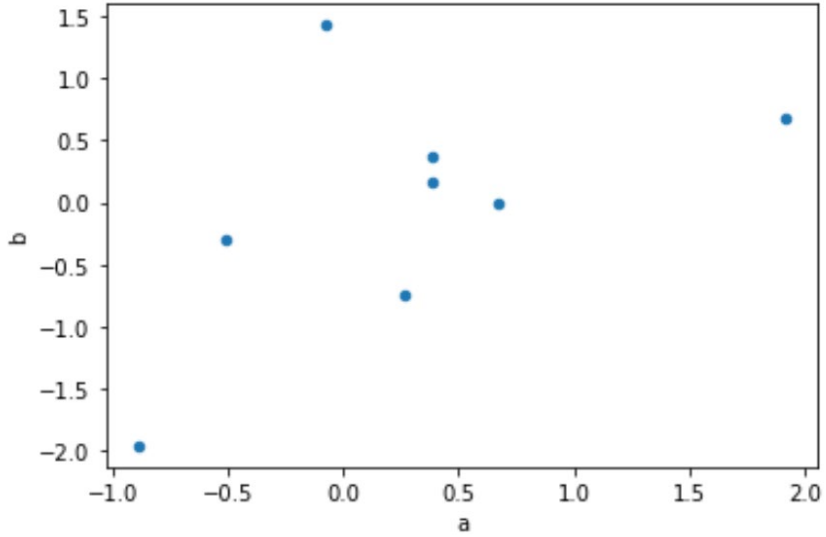
```
df
```

	a	b	c
2000-01-01	0.387906	0.165505	-0.296642
2000-01-02	0.675343	-0.012634	-0.946212
2000-01-03	-0.510775	-0.301052	-0.072443
2000-01-04	1.918426	0.679467	-0.840432
2000-01-05	-0.888401	-1.960367	0.568562
2000-01-06	0.266800	-0.738583	-0.915223
2000-01-07	-0.079060	1.429034	-0.038778
2000-01-08	0.387059	0.366470	0.222605

```
df.plot()
plt.show()
```




```
df.plot(kind = 'scatter', x = 'a', y = 'b')
plt.show()
```



Aggregating Statistics

- Different statistics are available and can be applied to columns with numerical data.
- Operations in general exclude missing data (NaN) and operate across rows by default.

```
df
```

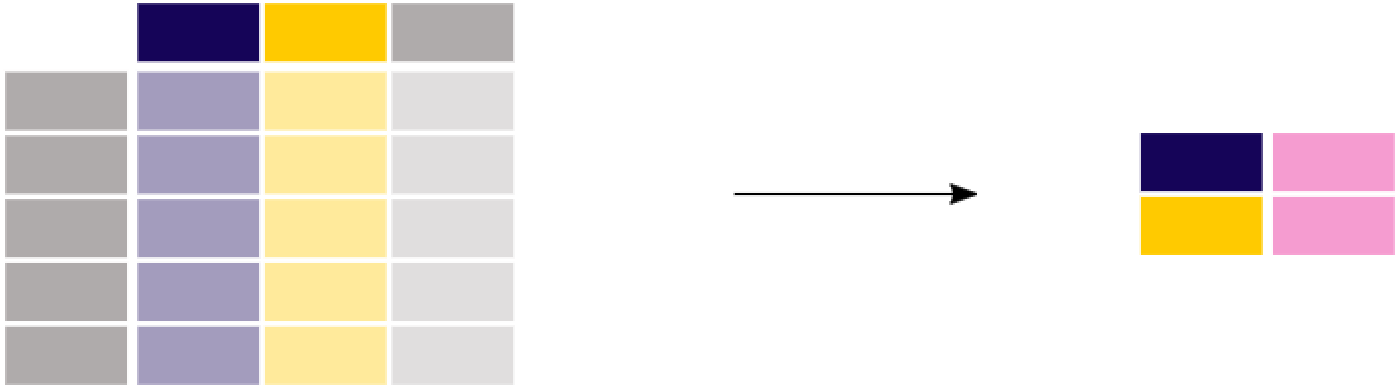
	a	b	c
2000-01-01	0.387906	0.165505	-0.296642
2000-01-02	0.675343	-0.012634	-0.946212
2000-01-03	-0.510775	-0.301052	-0.072443
2000-01-04	1.918426	0.679467	-0.840432
2000-01-05	-0.888401	-1.960367	0.568562
2000-01-06	0.266800	-0.738583	-0.915223
2000-01-07	-0.079060	1.429034	-0.038778
2000-01-08	0.387059	0.366470	0.222605

```
df['a'].mean()
```

0.26966233534457135

Aggregating Statistics

- The statistic applied to multiple columns of a DataFrame is calculated for each numeric column.
  - The selection of two columns returns a DataFrame



```
df[['a', 'b']].median()
```

```
a      0.326929
b      0.076436
dtype: float64
```

Aggregating Statistics

- The aggregating statistic can be calculated for multiple columns at the same time.
  - describe()

```
df[['a', 'b']].describe()
```

	a	b
count	8.000000	8.000000
mean	0.269662	-0.046520
std	0.843680	1.009068
min	-0.888401	-1.960367
25%	-0.186988	-0.410435
50%	0.326929	0.076436
75%	0.459765	0.444719
max	1.918426	1.429034

Aggregating Statistics

- Aggregating statistics grouped by category

```
df = pd.DataFrame({'a':[1,2,1,2,1,2,1,3,2,3],'b':[7,3,6,3,5,9,0,4,1,2]})
```

```
df
```

	a	b
0	1	7
1	2	3
2	1	6
3	2	3
4	1	5
5	2	9
6	1	0
7	3	4
8	2	1
9	3	2

```
df[['a', 'b']].groupby('a').mean()
```

	b
a	
1	4.5
2	4.0
3	3.0

Extracurricular Reading Materials

- 10 minutes to pandas
  - https://pandas.pydata.org/docs/user_guide/10min.html
- Getting started tutorials
  - https://pandas.pydata.org/docs/getting_started/intro_tutorials/index.html
- Advanced indexing
  - https://pandas.pydata.org/docs/user_guide/advanced.html
- Pandas tutorial for Data Science
  - https://towardsai.net/p/data-science/pandas-complete-tutorial-for-data-science-in-2022

Thank You!