Practical Business Python

Lecture 5: Getting Started with pandas.

Iegor Vyshnevskyi

Woosong University

October 12, 2023

Agenda

1. Pandas Intro
2. DataFrame
3. Essential Functionality
4. Summarizing and Computing Descriptive Statistics
5. In-class Assignment

1. Pandas Intro

Intro

Pandas is a widely used open-source Python library primarily designed for data manipulation and analysis. It provides a powerful and efficient data structure called DataFrame, which is akin to an in-memory 2D table, allowing for easy handling of structured data.

While Pandas adopts many coding idioms from NumPy, the biggest difference is that pandas is designed for working with tabular or heterogeneous data. NumPy, by contrast, is best suited for working with homogeneously typed numerical array data.

Since becoming an open source project in 2010, Pandas has matured into a quite large library that's applicable in a broad set of real-world use cases.

Source: Wes McKinney (2022)

Intro

Key features of Python Pandas:

- DataFrame:
  - The core data structure in pandas is the DataFrame, which represents tabular data with rows and columns, much like a spreadsheet or SQL table.
  - It allows for storing data of different types in columns and provides labeled axes (rows and columns) for easy indexing and alignment.
- Data Structures:
  - Besides DataFrame, pandas provides other data structures like Series (1D labeled array) and Panel (deprecated in newer versions), though DataFrame is the most widely used.
- Data Input and Output:
  - Pandas supports various file formats for input and output, including CSV, Excel, SQL databases, JSON, HTML, and more. It enables reading data from these sources into DataFrames and writing DataFrames to these formats.
- Data Cleaning:
  - Pandas offers functions to handle missing data, duplicate data, and inconsistent data, making data cleaning and preprocessing more efficient.
  - Methods like dropna(), fillna(), and duplicated() are used to handle missing values, fill nulls, and detect duplicates, respectively.
- Data Manipulation:
  - Pandas enables various operations on data, including merging and joining, reshaping and pivoting, slicing and indexing, and grouping and aggregating data.
  - Functions like merge(), concat(), pivot_table(), groupby(), and apply() are commonly used for these operations.

Intro

Key features of Python Pandas:

- Data Analysis:
  - It provides functionalities for descriptive statistics, correlation analysis, time series analysis, and more.
  - Functions like describe(), corr(), resample(), and rolling() aid in analyzing and summarizing data.
- Data Visualization Integration:
  - While not a core part of pandas, it integrates well with data visualization libraries like Matplotlib and Seaborn to create insightful plots and visualizations from DataFrames.
- Performance and Efficiency:
  - Pandas is designed to be fast and efficient. It's capable of handling large datasets efficiently.
- Multi-indexing:
  - Pandas supports multi-indexing, allowing for indexing by multiple criteria and levels. This is especially useful for handling complex, hierarchical data.
- Time Series Data:
  - Pandas has specialized support for time series data, making it convenient to work with date and time-related operations, including resampling, time shifting, and time zone handling.

Application

For most data analysis applications, the main **Pandas** areas of functionality are:

- Data Preprocessing and Cleaning:
  - Pandas is crucial in preparing data for analysis by handling missing values, transforming variables, encoding categorical data, and dealing with outliers. It plays a key role in ensuring data is in a suitable format for analysis.
- Exploratory Data Analysis (EDA):
  - Before diving into complex modeling, analysts use Pandas for initial exploration of the dataset. This includes generating summary statistics, creating visualizations, and understanding the structure and characteristics of the data.
- Time Series Analysis:
  - Pandas excels in handling time series data, making it a go-to tool for analyzing temporal data, trend analysis, forecasting, and seasonal pattern detection. It provides easy-to-use methods for resampling, shifting, and windowing time series.
- Statistical Analysis and Modeling:
  - Pandas is used for preparing data for statistical analysis, hypothesis testing, and model building. It helps in creating feature matrices, splitting datasets, and supporting various statistical tests and models.
- Business and Financial Analysis:
  - Professionals in finance and business use Pandas to analyze financial data, stock market trends, portfolio management, and risk assessment. It's crucial for calculating financial indicators, returns, and for financial modeling.

2. DataFrame

Our First Pandas DateFrame

A *DataFrame* represents a rectangular table of data and contains an ordered, named collection of columns, each of which can be a different value type (numeric, string, Boolean, etc.).

The DataFrame has both a row and column index; it can be thought of as a dictionary of Series all sharing the same index.

There are many ways to construct a DataFrame, though one of the most common is from a dictionary of equal-length lists or NumPy arrays:

```python
# Importing libraries
import numpy as np
import pandas as pd
from pandas import DataFrame # we use DataFrame a lot, so let's import it directly

# Creating a DataFrame
data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
        "year": [2000, 2001, 2002, 2001, 2002, 2003],
        "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
frame
```

Actual Script

Output

```
# Creating a DataFrame

data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
        "year": [2000, 2001, 2002, 2001, 2002, 2003],
        "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}

frame = pd.DataFrame(data)
frame
 state  year  pop
  Ohio  2000  1.5
  Ohio  2001  1.7
  Ohio  2002  3.6
Nevada  2001  2.4
Nevada  2002  2.9
Nevada  2003  3.2
```

Source: Wes McKinney (2022)

Our First Pandas DateFrame

Let's do some basic dataframe observation.

- For large DataFrames, the *head* method selects only the first five rows
- Similarly, *tail* returns the last five rows.

```
# Observe the first 5 rows
frame.head()

# Observe the last 5 rows
frame.tail()
```

```
>>> frame.head()
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
>>> # Observe the last 5 rows
>>>
>>> frame.tail()
    state  year  pop
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

Source: Wes McKinney (2022)

Our First Pandas DateFrame

Let's do some basic dataframe manipulation.

- If we specify a sequence of columns, the DataFrame's columns will be arranged in that order.
- If we pass a column that isn't contained in the dictionary, it will appear with missing values in the result.
- We check columns' names.

```
# Order the columns
pd.DataFrame(data, columns=["year", "state", "pop"])

# Create a new column
frame2 = pd.DataFrame(data, columns=["year", "state", "pop", "debt"])
frame2
frame2.columns # get the column names
```

```
>>> pd.DataFrame(data, columns=["year", "state", "pop"])
   year   state  pop
0  2000    Ohio  1.5
1  2001    Ohio  1.7
2  2002    Ohio  3.6
3  2001  Nevada  2.4
4  2002  Nevada  2.9
5  2003  Nevada  3.2
>>> # Create a new column
>>>
>>> frame2 = pd.DataFrame(data, columns=["year", "state", "pop", "debt"])
>>> frame2
   year   state  pop debt
0  2000    Ohio  1.5  NaN
1  2001    Ohio  1.7  NaN
2  2002    Ohio  3.6  NaN
3  2001  Nevada  2.4  NaN
4  2002  Nevada  2.9  NaN
5  2003  Nevada  3.2  NaN
>>> frame2.columns # get the column names
Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

Source: Wes McKinney (2022)

Our First Pandas DateFrame

Let's do some basic dataframe manipulation.

- A column in a DataFrame can be retrieved as a Series either by dictionary-like notation or by using the dot attribute notation.
- Rows can also be retrieved by position or name with the special *iloc* and *loc* attributes

```
# Get a column
frame2["state"]
frame2.year

# Get a row
frame2.loc[1]
frame2.iloc[2]
```

```
>>> frame2["state"]
0      Ohio
1      Ohio
2      Ohio
3    Nevada
4    Nevada
5    Nevada
Name: state, dtype: object
>>> frame2.year
0    2000
1    2001
2    2002
3    2001
4    2002
5    2003
Name: year, dtype: int64
```

```
>>> frame2.loc[1]
year     2001
state    Ohio
pop       1.7
debt      NaN
Name: 1, dtype: object
>>> frame2.iloc[2]
year     2002
state    Ohio
pop       3.6
debt      NaN
Name: 2, dtype: object
```

Source: Wes McKinney (2022)

Our First Pandas DateFrame

Let's do some basic dataframe manipulation.

- Columns can be modified by assignment. For example, the empty debt column could be assigned a scalar value or an array of values.
- When we are assigning lists or arrays to a column, the value's length must match the length of the DataFrame. If we assign a Series, its labels will be realigned exactly to the DataFrame's index, inserting missing values in any index values not present.

```
# Modify a column
frame2["debt"] = 16.5
frame2

frame2["debt"] = np.arange(6.)

val = pd.Series([-1.2, -1.5, -1.7], index=[2, 4, 5])
frame2["debt"] = val
frame2
```

```
>>> frame2["debt"] = 16.5
>>> frame2
   year   state  pop  debt
0  2000    Ohio  1.5  16.5
1  2001    Ohio  1.7  16.5
2  2002    Ohio  3.6  16.5
3  2001  Nevada  2.4  16.5
4  2002  Nevada  2.9  16.5
5  2003  Nevada  3.2  16.5
>>> frame2["debt"] = np.arange(6.)
>>> val = pd.Series([-1.2, -1.5, -1.7], index=[2, 4, 5])
>>> frame2["debt"] = val
>>> frame2
   year   state  pop  debt
0  2000    Ohio  1.5   NaN
1  2001    Ohio  1.7   NaN
2  2002    Ohio  3.6  -1.2
3  2001  Nevada  2.4   NaN
4  2002  Nevada  2.9  -1.5
5  2003  Nevada  3.2  -1.7
```

Source: Wes McKinney (2022)

Our First Pandas DateFrame

Let’s do some basic dataframe manipulation.

- Assigning a column that doesn’t exist will create a new column.
- The *del* keyword will delete columns like with a dictionary. As an example, we first add a new column of Boolean values where the *state* column equals "Ohio":

```
# Delete a column
frame2["eastern"] = frame2["state"] == "Ohio"
frame2

del frame2["eastern"]
frame2.columns
```

```
>>> frame2["eastern"] = frame2["state"] == "Ohio"
>>> frame2
   year   state  pop  debt  eastern
0  2000    Ohio  1.5   NaN     True
1  2001    Ohio  1.7   NaN     True
2  2002    Ohio  3.6  -1.2     True
3  2001  Nevada  2.4   NaN    False
4  2002  Nevada  2.9  -1.5    False
5  2003  Nevada  3.2  -1.7    False
>>> del frame2["eastern"]
>>> frame2.columns
Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

Source: Wes McKinney (2022)

Our First Pandas DateFrame

Another common form of data is a nested dictionary of dictionaries.

- If the nested dictionary is passed to the DataFrame, pandas will interpret the outer dictionary keys as the columns, and the inner keys as the row indices.
- We can transpose the DataFrame (swap rows and columns) with similar syntax to a NumPy array.

```python
# Create a DataFrame from a nested dict of dicts
populations = {"Nevada": {2001: 2.4, 2002: 2.9},
               "Ohio": {2000: 1.5, 2001: 1.7, 2002: 3.6}}
frame3 = pd.DataFrame(populations)
frame3

# Transpose the DataFrame
frame3.T
```

```
>>> populations = {"Nevada": {2001: 2.4, 2002: 2.9},
...                "Ohio": {2000: 1.5, 2001: 1.7, 2002: 3.6}}
>>>
>>> frame3 = pd.DataFrame(populations)
>>> frame3
      Nevada  Ohio
2001     2.4   1.7
2002     2.9   3.6
2000     NaN   1.5
>>> # Transpose the DataFrame
>>>
>>> frame3.T
        2001  2002  2000
Nevada   2.4   2.9   NaN
Ohio     1.7   3.6   1.5
```

Source: Wes McKinney (2022)

Index Objects

Pandas's Index objects are responsible for holding the axis labels (including a DataFrame's column names) and other metadata (like the axis name or names). Any array or other sequence of labels we use when constructing DataFrame is internally converted to an Index.

```
# Recall frame3
frame3
frame3.columns
# Indexing
"Ohio" in frame3.columns
2003 in frame3.index
```

```
>>> frame3
      Nevada  Ohio
2001     2.4   1.7
2002     2.9   3.6
2000     NaN   1.5
>>> frame3.columns
Index(['Nevada', 'Ohio'], dtype='object')
>>> # Indexing
>>>
>>> "Ohio" in frame3.columns
True
>>> 2003 in frame3.index
False
```

Source: Wes McKinney (2022)

3. Essential Functionality

Reindexing

An important method on pandas objects is *reindex*, which means to create a new object with the values rearranged to align with the new index.

- With DataFrame, *reindex* can alter the (row) index, columns, or both. When passed only a sequence, it reindexes the rows.
- The columns can be reindexed with the columns keyword (Because "Ohio" was not in states, the data for that column is dropped from the result).

```
# Create a DataFrame from a dict of Series
frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
   index=["a", "c", "d"],
   columns=["Ohio", "Texas", "California"])
frame

# Row reindexing
frame2 = frame.reindex(index=["a", "b", "c", "d"])
frame2

# Columns can be reindexed with the columns keyword
states = ["Texas", "Utah", "California"]
frame.reindex(columns=states)
# Another way to do it
frame.reindex(states, axis="columns")
```

```
>>> frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
...     index=["a", "c", "d"],
...     columns=["Ohio", "Texas", "California"])
>>>
>>> frame
   Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8
>>> # Row reindexing
>>>
>>> frame2 = frame.reindex(index=["a", "b", "c", "d"])
>>> frame2
   Ohio  Texas  California
a   0.0    1.0         2.0
b   NaN    NaN         NaN
c   3.0    4.0         5.0
d   6.0    7.0         8.0
```

```
>>> # Columns can be reindexed with the columns keyword
>>>
>>> states = ["Texas", "Utah", "California"]
>>> frame.reindex(columns=states)
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
>>> # Another way to do it
>>>
>>> frame.reindex(states, axis="columns")
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
```

Source: Wes McKinney (2022)

Dropping Entries from an Axis

Dropping one or more entries from an axis is simple if you already have an index array or list without those entries, since you can use the reindex method or .loc-based indexing. As that can require a bit of munging and set logic, the drop method will return a new object with the indicated value or values deleted from an axis.

- With DataFrame, index values can be deleted from either axis.
- Calling *drop* with a sequence of labels will drop values from the row labels (axis 0).
- To drop labels from the columns, instead use the *columns* keyword.
- You can also drop values from the columns by passing *axis=1* (which is like NumPy) or *axis="columns“.*

```
# Dropping entries from an axis
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
    index=["Ohio", "Colorado", "Utah", "New York"],
    columns=["one", "two", "three", "four"])
data

# Drop rows
data.drop(index=["Colorado", "Ohio"])
# Drop columns
data.drop(columns=["two"])
#
data.drop("two", axis=1)
data.drop(["two", "four"], axis="columns")
```

```
>>> # Drop rows
>>>
>>> data.drop(index=["Colorado", "Ohio"])
          one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15
>>> # Drop columns
>>>
>>> data.drop(columns=["two"])
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15
```

```
>>> #
>>>
>>> data.drop("two", axis=1)
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15
>>> data.drop(["two", "four"], axis="columns")
          one  three
Ohio        0      2
Colorado    4      6
Utah        8     10
New York   12     14
```

Source: Wes McKinney (2022)

Indexing, Selection, and Filtering

Indexing into a DataFrame retrieves one or more columns either with a single value or sequence:

```python
# Indexing, selection, and filtering
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
    index=["Ohio", "Colorado", "Utah", "New York"],
    columns=["one", "two", "three", "four"])
data

data["two"] # get a column
data[["three", "one"]] # get multiple columns
data[:2] # get the first two rows
data[data["three"] > 5] # get rows where the value in column "three" is greater than 5
data < 5 # get a boolean DataFrame
data[data < 5] = 0 # set values less than 5 to 0
data
```

```
>>> data
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
>>> data["two"] # get a column
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int32
>>> data[["three", "one"]] # get multiple columns
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
New York     14   12
>>> data[:2] # get the first two rows
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
```

```
>>> data[data["three"] > 5] # get rows where the value in column "three" is greater than 5
          one  two  three  four
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
>>> data < 5 # get a boolean DataFrame
            one    two  three   four
Ohio       True   True   True   True
Colorado   True  False  False  False
Utah      False  False  False  False
New York  False  False  False  False
>>> data[data < 5] = 0 # set values less than 5 to 0
>>> data
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

Source: Wes McKinney (2022)

Selection on DataFrame with loc and iloc

DataFrame has special attributes *loc* and *iloc* for label-based and integer-based indexing, respectively. Since DataFrame is two-dimensional, we can select a subset of the rows and columns with NumPy-like notation using either axis labels (*loc*) or integers (*iloc*).

```python
# Selecting with loc and iloc
data.loc["Colorado"]
data.loc["Colorado", ["two", "three"]]
data.iloc[2, [3, 0, 1]]
data.iloc[2]
data.iloc[[1, 2], [3, 0, 1]]
data.loc[: "Utah", "two"]
data.iloc[:, :3][data.three > 5]
data.loc[data.three >= 2]
```

```
>>> data.loc["Colorado"]
one      0
two      5
three    6
four     7
Name: Colorado, dtype: int32
>>> data.loc["Colorado", ["two", "three"]]
two      5
three    6
Name: Colorado, dtype: int32
>>> data.iloc[2, [3, 0, 1]]
four    11
one      8
two      9
Name: Utah, dtype: int32
>>> data.iloc[2]
one       8
two       9
three    10
four     11
Name: Utah, dtype: int32
```

```
>>> data.iloc[[1, 2], [3, 0, 1]]
          four  one  two
Colorado     7    0    5
Utah        11    8    9
>>> data.loc[: "Utah", "two"]
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int32
>>> data.iloc[:, :3][data.three > 5]
          one  two  three
Colorado    0    5      6
Utah        8    9     10
New York   12   13     14
>>> data.loc[data.three >= 2]
          one  two  three  four
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
>>>
```

Source: Wes McKinney (2022)

Arithmetic and Data Alignment

Alignment is performed on both rows and columns.

- Adding two DataFrames returns a DataFrame with index and columns that are the unions of the ones in each DataFrame.
- Since the "*c*" and "*e*" columns are not found in both DataFrame objects, they appear as missing in the result. The same holds for the rows with labels that are not common to both objects.

```python
# Arithmetic and data alignment
df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list("bcd"),
    index=["Ohio", "Texas", "Colorado"])

df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list("bde"),
    index=["Utah", "Ohio", "Texas", "Oregon"])

df1
df2

# Add two DataFrames
df1 + df2
```

```
>>> df1
            b    c    d
Ohio      0.0  1.0  2.0
Texas     3.0  4.0  5.0
Colorado  6.0  7.0  8.0
>>> df2
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0
>>> # Add two DataFrames
>>>
>>> df1 + df2
            b   c     d   e
Colorado  NaN NaN   NaN NaN
Ohio      3.0 NaN   6.0 NaN
Oregon    NaN NaN   NaN NaN
Texas     9.0 NaN  12.0 NaN
Utah      NaN NaN   NaN NaN
```

Source: Wes McKinney (2022)

Arithmetic and Data Alignment (fill values)

In arithmetic operations between differently indexed objects, you might want to fill with a special value, like 0, when an axis label is found in one object but not the other.

- Here is an example where we set a particular value to NA (null) by assigning *np.nan* to it.

```
# Arithmetic methods with fill values
df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)), columns=list("abcd"))
df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)), columns=list("abcde"))

df2.loc[1, "b"] = np.nan # add a missing value

df1 + df2

# Fill missing values with 0
df1.add(df2, fill_value=0)
```

```
>>> df1 + df2
      a     b     c     d   e
0   0.0   2.0   4.0   6.0 NaN
1   9.0   NaN  13.0  15.0 NaN
2  18.0  20.0  22.0  24.0 NaN
3   NaN   NaN   NaN   NaN NaN
>>> # Fill missing values with 0
>>>
>>> df1.add(df2, fill_value=0)
      a     b     c     d     e
0   0.0   2.0   4.0   6.0   4.0
1   9.0   5.0  13.0  15.0   9.0
2  18.0  20.0  22.0  24.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

Source: Wes McKinney (2022)

Function Application and Mapping

NumPy ufuncs (element-wise array methods) also work with pandas objects.

- Another frequent operation is applying a function on one-dimensional arrays to each column or row. DataFrame's *apply* method does exactly this.
- Here the function *f*, which computes the difference between the maximum and minimum of a Series, is invoked once on each column in frame. The result is a Series having the columns of frame as its index.
- If we pass *axis="columns"* to apply, the function will be invoked once per row instead. A helpful way to think about this is as "apply across the columns“.

```python
# Function application and mapping
frame = pd.DataFrame(np.random.standard_normal((4, 3)),
    columns=list("bde"),
    index=["Utah", "Ohio", "Texas", "Oregon"])

frame

np.abs(frame) # apply a function to each column

def f1(x):
    return x.max() - x.min() # define a function

frame.apply(f1) # apply the function to each column
frame.apply(f1, axis="columns") # apply the function to each row

def f2(x):
    return pd.Series([x.min(), x.max()], index=["min", "max"]) # define a function

frame.apply(f2)
```

Source: *Wes McKinney (2022)*

```
>>> np.abs(frame) # apply a function to each column
               b         d         e
Utah    0.213351  0.511324  0.191095
Ohio    0.673054  1.402477  0.714909
Texas   0.861411  1.629442  1.603999
Oregon  2.871411  0.362244  0.480223
>>> def f1(x):
...     return x.max() - x.min() # define a function
...
>>> frame.apply(f1) # apply the function to each column
b    3.732822
d    3.031919
e    2.318908
dtype: float64
>>> frame.apply(f1, axis="columns") # apply the function to each row
Utah      0.702419
Ohio      2.117386
Texas     3.233441
Oregon    3.351634
dtype: float64
>>> def f2(x):
...     return pd.Series([x.min(), x.max()], index=["min", "max"]) # define a function
...
>>> frame.apply(f2)
            b         d         e
min -0.861411 -1.402477 -1.603999
max  2.871411  1.629442  0.714909
```

Sorting and Ranking

Sorting a dataset by some criterion is another important built-in operation. To sort lexicographically by row or column label, use the *sort_index* method, which returns a new, sorted object.

- With a DataFrame, you can sort by index on either axis.
- The data is sorted in ascending order by default but can be sorted in descending order.

```
# Sorting and ranking
frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
    index=["three", "one"],
    columns=["d", "a", "b", "c"])
frame

frame.sort_index() # sort by row index
frame.sort_index(axis="columns") # sort by columns
frame.sort_index(axis="columns", ascending=False) # sort by columns in descending order
```

```
>>> frame
       d  a  b  c
three  0  1  2  3
one    4  5  6  7
>>> frame.sort_index() # sort by row index
       d  a  b  c
one    4  5  6  7
three  0  1  2  3
>>> frame.sort_index(axis="columns") # sort by columns
       a  b  c  d
three  1  2  3  0
one    5  6  7  4
>>> frame.sort_index(axis="columns", ascending=False) # sort by columns in descending order
       d  c  b  a
three  0  3  2  1
one    4  7  6  5
```

Source: Wes McKinney (2022)

Sorting and Ranking

Ranking assigns ranks from one through the number of valid data points in an array, starting from the lowest value. The *rank* methods for Series and DataFrame are the place to look; by default, *rank* breaks ties by assigning each group the mean rank.

```
# Ranking
frame = pd.DataFrame({"b": [4.3, 7, -3, 2], "a": [0, 1, 0, 1],
    "c": [-2, 5, 8, -2.5]})
frame

frame.rank() # rank by row
frame.rank(axis="columns") # rank by column
```

```
>>> frame
     b  a    c
0  4.3  0 -2.0
1  7.0  1  5.0
2 -3.0  0  8.0
3  2.0  1 -2.5
>>> frame.rank() # rank by row
     b    a    c
0  3.0  1.5  2.0
1  4.0  3.5  3.0
2  1.0  1.5  4.0
3  2.0  3.5  1.0
>>> frame.rank(axis="columns") # rank by column
     b    a    c
0  3.0  2.0  1.0
1  3.0  1.0  2.0
2  1.0  2.0  3.0
3  3.0  2.0  1.0
```

Source: Wes McKinney (2022)

4. Summarizing and Computing Descriptive Statistics

Mathematical and Statistical Methods

Pandas objects are equipped with a set of common mathematical and statistical methods. Most of these fall into the category of reductions or summary statistics, methods that extract a single value (like the sum or mean) from a Series of values from the rows or columns of a DataFrame.

Compared with the similar methods found on NumPy arrays, they have built-in handling for missing data.

```python
# Summarizing and computing descriptive statistics
df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],
    [np.nan, np.nan], [0.75, -1.3]],
    index=["a", "b", "c", "d"],
    columns=["one", "two"])
df

df.sum() # sum by column
df.sum(axis="columns") # sum by row
df.mean(axis="columns", skipna=False) # mean by row, including missing values
df.idxmax() # get the index of the maximum value in each column
df.cumsum() # cumulative sum by column
df.describe() # get summary statistics
```

```
>>> df
    one  two
a  1.40  NaN
b  7.10 -4.5
c   NaN  NaN
d  0.75 -1.3
>>> df.sum() # sum by column
one    9.25
two   -5.80
dtype: float64
>>> df.sum(axis="columns") # sum by row
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

```
>>> df.idxmax() # get the index of the maximum value in each column
one    b
two    d
dtype: object
>>> df.cumsum() # cumulative sum by column
    one  two
a  1.40  NaN
b  8.50 -4.5
c   NaN  NaN
d  9.25 -5.8
>>> df.describe() # get summary statistics
            one       two
count  3.000000  2.000000
mean   3.083333 -2.900000
std    3.493685  2.262742
min    0.750000 -4.500000
25%    1.075000 -3.700000
50%    1.400000 -2.900000
75%    4.250000 -2.100000
max    7.100000 -1.300000
```

Source: Wes McKinney (2022)

5. In-class Assignment