Getting Started with pandas

Prof. Gheith Abandah

Getting Started with pandas

- YouTube Video from **Python Programmer**

What is Pandas? Why and How to Use Pandas in Python

https://youtu.be/dcqPhpY7tWk

Reference

- **Chapter 5**
- Wes McKinney, **Python for Data Analysis**: Data Wrangling with Pandas, NumPy, and IPython, O'Reilly Media, 2nd Edition, 2018.
  - Material: https://github.com/wesm/pydata-book

Outline

5.1 Introduction to pandas Data Structures

5.2 Essential Functionality

5.3 Summarizing and Computing Descriptive Statistics

Pandas: Labeled Column-Oriented Data

- Pandas is high-performance, easy-to-use **data structures** and **data analysis tools**. It contains:
  - A set of labeled array data structures (**Series** and **DataFrame**)
  - **Index objects** enabling both simple axis indexing and multi-level / hierarchical axis indexing
  - **Input/Output tools**: loading tabular data from flat files (CSV, delimited, Excel 2003), and saving and loading pandas objects
- **Website**: https://pandas.pydata.org/
- Also, check the **tutorial** on Learn Python: https://www.learnpython.org/en/Pandas_Basics

Outline

5.1 Introduction to pandas Data Structures

5.2 Essential Functionality

5.3 Summarizing and Computing Descriptive Statistics

- Series
- DataFrame
- Index Objects

Series

- **One-dimensional** array-like object containing a **sequence of values** (of similar **types** to NumPy types) and an associated array of data labels, called its **index**.
- Has `.values` and `.index` attributes.

```
obj = pd.Series([4, 7, -5, 3])
obj
0  4
1  7
2 -5
3  3
dtype: int64
obj.values
array([ 4, 7, -5, 3])
obj.index # like range(4)
RangeIndex(start=0, stop=4, step=1)
```

May not show in following slides

Index object

Series

- You can **specify** the **index**
  - at **creation**
  - or by **assignment**.

```
obj = pd.Series([7, -5, 3],
                index=['b', 'a', 'c'])
obj
b  7
a -5
c  3
```

- You can **use labels** in the index when **selecting** single values or a set of values.

```
obj.index = ['b', 'a', 'c']
obj['a'] = 6
obj[['c', 'a']]
c 3
a 6
```

Series

- Accepts **functions** and **operations like NumPy**:
  - **Filtering** with a Boolean array
  - **Scalar multiplication**
  - Applying **math functions**

```
obj[obj > 5]
b   7
a   6
```

```
obj * 2
b   14
a   12
c    6
```

```
np.square(obj)
b   49
a   36
c    9
```

```
obj
b   7
a   6
c   3
```

Series

- You can check labels similar to dictionaries using the **in** operator.
- Can convert **dictionary to Series**.
- Can **rearrange** with new index.
- Support methods like **.isnull**.

```
obj
b   7
a   6
c   3
```

```
'b' in obj
True
sdata = {'Ohio': 35000, 'Texas': 71000}
obj3 = pd.Series(sdata)
states = ['California', 'Texas']
obj4 = pd.Series(sdata, index=states)
obj4
California       NaN
Texas        71000.0

pd.isnull(obj4)
California    True
Texas        False
```

DataFrame

- **Table** of data containing an ordered collection of **columns**, each of which can be a **different** value **type**.
- Has both a **row** and **column index**; it can be thought of as a **dict of Series** all sharing the same index.
- Supports `.head` and `.tail`.

```
data = {'state': ['Ohio', 'Nevada'],
        'year': [2002, 2001],
        'pop': [3.6, 2.4]}
frame = pd.DataFrame(data)
frame.head()
   pop   state  year
0  3.6    Ohio  2002
1  2.4  Nevada  2001
```

DataFrame

- You can specify **columns** and **index**.

- You can check the **columns** labels or **index** using the corresponding attributes.

```
arr = np.arange(4).reshape(2, 2)
df = pd.DataFrame(arr,
        columns=['one', 'two'],
        index=['a', 'b'])
df
```

```
   one  two
a    0    1
b    2    3
```

```
df.columns
```

```
Index(['one', 'two'], dtype='object')
```

DataFrame

- You can access a column or a value by `[]` or `label`.
- When used as label, the name should be **valid Python name**.
- You can access rows by `.loc` method.

```
df
     one  two
a      0    1
b      2    3
```

```
df['one']
a     0
b     2
df.two['b']
3
df.loc['b']
one     2
two     3
```

Series

DataFrame

```
df
    one  two
a     0    1
b     2    3
```

- Columns could be **assigned** scalars, vectors, or Series.
- Adding **new columns** is easy.
- Use **del** to delete columns.

```
val = pd.Series([-1.2, -1.5],
                index=['b', 'a'])
df['one'] = val
df['three'] = 3
df
    one  two  three
a -1.5    1      3
b -1.2    3      3
del df['one']
df.columns
Index(['two', 'three'], dtype='object')
```

Index Objects

```
df
   two  three
a    1      3
b    3      3
```

- **Immutable** objects that hold axis labels.
- An **Index** behaves like a fixed-size set, but it allows duplicates.

```
df.index
Index(['a', 'b'], dtype='object')
labels = pd.Index(np.arange(3))
obj2 = pd.Series([1.5, -2.5, 0],
                 index=labels)
obj2
0  1.5
1 -2.5
2  0.0
'two' in df.columns
True
```

Outline

5.1 Introduction to pandas Data Structures

5.2 Essential Functionality

5.3 Summarizing and Computing Descriptive Statistics

- Reindexing
- Dropping Entries from an Axis
- Indexing, Selection, and Filtering
- Arithmetic and Data Alignment
- Function Application and Mapping
- Sorting and Ranking
- Axis Indexes with Duplicate Labels

Reindexing

- If you want to rearrange the data according to a new index, use **reindex**.

- The columns can be reindexed with the **columns** keyword.

```
obj = pd.Series([7.2, -5.3, 3.6],
                index=['b', 'a', 'c'])
obj
b  7.2
a -5.3
c  3.6
obj2 = obj.reindex(['a', 'b', 'e'])
obj2
a -5.3
b  7.2
e  NaN
df.reindex(columns=new_index)
```

Dropping Entries from an Axis

- If you don’t want some data, you can drop rows or columns.
  - inplace
  - or in a **new object**
- You can drop **multiple** items or **one** item.

```
data =
pd.DataFrame(np.arange(9).reshape((3, 3)),
         index=['Ohio', 'Colorado', 'Utah'],
         columns=['one', 'two', 'three'])
data
          one  two  three
Ohio        0    1      2
Colorado    3    4      5
Utah        6    7      9
data.drop(['Utah', 'Ohio'] , inplace=True)
data.drop('three', axis=1)
          one  two
Colorado    3    4
```

Indexing, Selection, and Filtering

- You can use **integers** to index **Series** as well as labels.

```
obj['b']

obj[1]

obj[['b', 'a', 'd']]

obj[[1, 3]]
```

- **Slicing** with **labels includes** the **endpoint**.

```
obj[2:4]

obj['b':'c'] = 5
```

Indexing, Selection, and Filtering

- With DataFrames, **[ ]** selects **column(s)**.
- But **slicing** and selection with a **Boolean array** selects **rows**.
- The Boolean array can be of the same DataFrame size.

```
data['two']
```

```
data[:2]
```

```
data[data['three'] > 5]
```

```
data[data < 5] = 0
```

Selection with loc and iloc

- DataFrame **indexing on the rows**: to select a subset of the rows and columns, use:
  - axis labels with `loc`
  - or integers with `iloc`.

```
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
```

```
data.loc['Colorado', ['two', 'three']]
two    5
three  6
data.iloc[2, [3, 0, 1]]
four 11
one    8
two    9
```

Arithmetic and Data Alignment

- When you are **adding objects**, if any index pairs are not the same, the respective index in the result will be the **union** of the index pairs.

```
s1
a   7.3
c  -2.5
d   3.4
e   1.5
```

```
s2
a  -2.1
c   3.6
e  -1.5
f   4.0
g   3.1
```

```
s1 + s2
a   5.2
c   1.1
d   NaN
e   0.0
f   NaN
g   NaN
```

Arithmetic and Data Alignment

- In the case of **DataFrame**, **alignment** is performed on **both** the **rows** and the **columns**.

```
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
```

```
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
```

```
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
```

Arithmetic methods with fill values

- You can **fill** with a special value, like `0`, when an axis label is found in one object but not the other using `add(fill_value=0)`.

```
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
```

```
df1.add(df2, fill_value=0)
            b    c     d     e
Colorado  6.0  7.0   8.0   NaN
Ohio      3.0  1.0   6.0   5.0
Oregon    9.0  NaN  10.0  11.0
Texas     9.0  4.0  12.0   8.0
Utah      0.0  NaN   1.0   2.0
```

Arithmetic methods with fill values

- Series and DataFrame **methods** for **arithmetic**.
- The method starting with the letter **r** has arguments flipped.
- Equivalent:

```
1 / df1
df1.rdiv(1)
```

Method	Description
`add, radd`	Methods for addition (+)
`sub, rsub`	Methods for subtraction (-)
`div, rdiv`	Methods for division (/)
`floordiv, rfloordiv`	Methods for floor division (//)
`mul, rmul`	Methods for multiplication (*)
`pow, rpow`	Methods for exponentiation (**)

Operations between DataFrame and Series

- **Arithmetic** between **DataFrame** and **Series** is possible.
- By default, arithmetic matches the index of the Series on the DataFrame's columns, **broadcasting down** the **rows**.

```
series = frame.iloc[0]
```

```
series
b   0.0
d   1.0
e   2.0
```

```
frame
          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
```

```
frame - series
          b    d    e
Utah    0.0  0.0  0.0
Ohio    3.0  3.0  3.0
Texas   6.0  6.0  6.0
Oregon  9.0  9.0  9.0
```

Operations between DataFrame and Series

- If you want to **broadcast** over the **columns**, matching on the rows, you have to use one of the arithmetic methods with **`axis='index'`**.

```
series3 = frame['d']
```

```
series3
Utah     1.0
Ohio     4.0
Texas    7.0
Oregon  10.0
```

```
frame
          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
```

```
frame.sub(series3, axis='index')
          b    d    e
Utah   -1.0  0.0  1.0
Ohio   -1.0  0.0  1.0
Texas  -1.0  0.0  1.0
Oregon -1.0  0.0  1.0
```

or 0

Function Application and Mapping

- **NumPy ufuncs** work with **pandas** objects.

```
frame = pd.DataFrame(randn(2, 3),
           columns=list('bde'),
           index=['Utah', 'Ohio'])
frame
             b         d         e
Utah -0.291993  0.085824 -0.222663
Ohio -1.473446  1.049407 -1.035874
 np.abs(frame)
             b         d         e
Utah  0.291993  0.085824  0.222663
Ohio  1.473446  1.049407  1.035874
```

Function Application and Mapping

- DataFrame’s **apply** method applies a **function** on **one-dimensional arrays** to each column
- or row.
- Element-wise Python functions:
  - **map** for Series
  - **applymap** for DataFrame

```
f = lambda x: x.max() - x.min()
frame.apply(f)
b     1.181453
d     0.963583
e     0.813211
```

```
frame
             b         d         e
Utah -0.291993  0.085824 -0.222663
Ohio -1.473446  1.049407 -1.035874
```

```
frame.apply(f, axis='columns')
Utah     0.377817
Ohio     2.522853
```

```
frame['e'].map(f2)
```

```
frame.applymap(f2)
```

Sorting and Ranking

- You can **sort** Series and DataFrame using **sort_index**.
- You can select the **axis** and **sort direction**.
- You can also sort by the **values** of one or multiple columns.

```
frame
      d a b c
three 0 1 2 3
one   4 5 6 7
```

```
frame.sort_index(axis=1,
                 ascending=False)
```

```
      d c b a
three 0 3 2 1
one   4 7 6 5
```

```
frame.sort_values(by=['a', 'b'])
```

Sorting and Ranking

- **Ranking** assigns ranks from **1** through the number of valid data points in an array.
- Use panda's **rank**.

Table 5-6. Tie-breaking methods with rank

Method	Description
'average'	Default: assign the average rank to each entry in the equal group
'min'	Use the minimum rank for the whole group
'max'	Use the maximum rank for the whole group
'first'	Assign ranks in the order the values appear in the data
'dense'	Like method='min', but ranks always increase by 1 in between groups rather than the number of equal elements in a group

```
obj = pd.Series([7, -5, 7, 4,
                 2, 0, 4])
obj.rank()
0   6.5
1   1.0
2   6.5
3   4.5
4   3.0
5   2.0
6   4.5
df.rank(ascending=False, method='max',
        axis='columns')
```

Has these familiar options.

Axis Indexes with Duplicate Labels

- Indexing a label in a **Series** usually returns a **scalar**.
- Indexing a label with multiple entries returns a **Series**.
- And returns a **DataFrame** from a DataFrame.

```
df = pd.DataFrame(randn(4, 3),
                  index=['a', 'a', 'b', 'b'])
df
          0         1         2
a 0.274992  0.228913  1.352917
a 0.886429 -2.001637 -0.371843
b 1.669025 -0.438570 -0.539741
b 0.476985  3.248944 -1.021228
df.loc['b']
          0         1         2
b 1.669025 -0.438570 -0.539741
b 0.476985  3.248944 -1.021228
```

Outline

5.1 Introduction to pandas Data Structures

5.2 Essential Functionality

5.3 Summarizing and Computing Descriptive Statistics

- Correlation and Covariance
- Unique Values, Value Counts, and Membership

5.3 Summarizing and Computing Descriptive Statistics

- pandas has **mathematical** and **statistical** methods.
- Most are **reductions** methods like the **sum** or **mean**.
- They have built-in **handling** for **missing data**.

```
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
    one  two
a  1.40  NaN
b  7.10 -4.5
c   NaN  NaN
d  0.75 -1.3
df.sum()
one   9.25
two  -5.80
```

```
df.sum(axis='columns')
a  1.40
b  2.60
c   NaN
d -0.55
```

5.3 Summarizing and Computing Descriptive Statistics

- Some return **index** like **`idxmax`**.

```
df.idxmax()
one b
two d
```

- Or **accumulations** like **`cumsum`**.

```
df.cumsum()
   one  two
a  1.40  NaN
b  8.50 -4.5
c   NaN  NaN
d  9.25 -5.8
```

```
df
   one  two
a 1.40  NaN
b 7.10 -4.5
c  NaN  NaN
d 0.75 -1.3
```

5.3 Summarizing and Computing Descriptive Statistics

- **describe** returns summary statistics.
- On **non-numeric** data, it produces **alternative summary** statistics.

```
df
   one  two
a 1.40  NaN
b 7.10 -4.5
c  NaN  NaN
d 0.75 -1.3
```

```
df.describe()
            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
obj = pd.Series(['a', 'a',
                 'b', 'c'] * 4)
obj.describe()
count   16
unique   3
top      a
freq     8
```

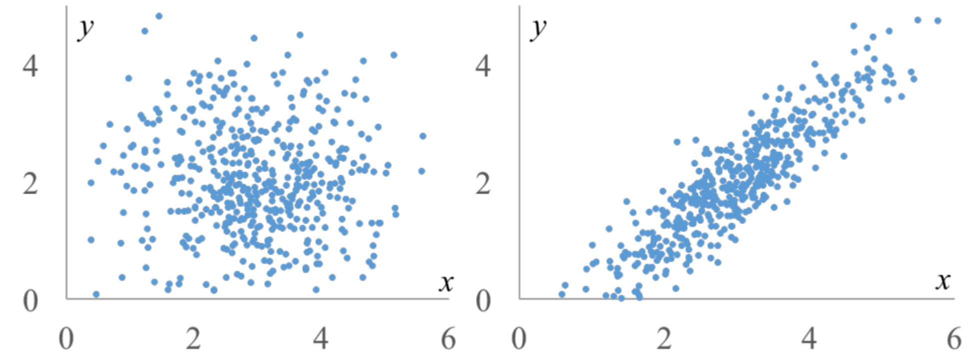
Descriptive and summary statistics

Method	Description
count	Number of non-NA values
describe	Compute set of summary statistics for Series or each DataFrame column
min, max	Compute minimum and maximum values
argmin, argmax	Compute index locations (integers) at which minimum or maximum value obtained, respectively
idxmin, idxmax	Compute index labels at which minimum or maximum value obtained, respectively
quantile	Compute sample quantile ranging from 0 to 1
sum	Sum of values
mean	Mean of values
median	Arithmetic median (50% quantile) of values
mad	Mean absolute deviation from mean value

Descriptive and summary statistics – cont.

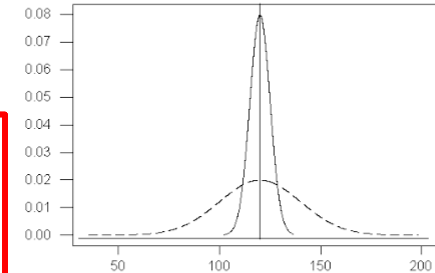
`prod`	Product of all values
`var`	Sample variance of values
`std`	Sample standard deviation of values
`skew`	Sample skewness (third moment) of values
`kurt`	Sample kurtosis (fourth moment) of values
`cumsum`	Cumulative sum of values
`cummin, cummax`	Cumulative minimum or maximum of values, respectively
`cumprod`	Cumulative product of values
`diff`	Compute first arithmetic difference (useful for time series)
`pct_change`	Compute percent changes

Correlation and Covariance



- **Covariance** is a quantitative measure of the extent to which the deviation of one variable from its mean matches the deviation of the other from its mean.
- **Correlation** between two random variables is the covariance of the two variables **normalized** by the variance of each variable.

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$



$$\mathrm{Var}(X) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

$$\mathrm{cov}(X,Y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - E(X))(y_i - E(Y)).$$

$$\mathrm{Corr}(X,Y) = \frac{\mathrm{Cov}(X,Y)}{\sqrt{\mathrm{Var}(X)\,\mathrm{Var}(Y)}}$$

Correlation and Covariance

- Case Study: **Yahoo! Finance**

```
conda install pandas-datareader

import pandas_datareader.data as web

all_data = {ticker: web.get_data_yahoo(ticker)
            for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']}

price = pd.DataFrame({ticker: data['Adj Close']
                      for ticker, data in all_data.items()})
```

Correlation and Covariance

```
price.tail(3)
                  AAPL         IBM        MSFT         GOOG
Date
2020-11-11  119.489998  117.199997  216.550003  1752.709961
2020-11-12  119.209999  114.500000  215.440002  1749.839966
2020-11-13  119.260002  116.849998  216.509995  1777.020020
```

- **Percent change**

```
returns = price.pct_change()
returns.tail(3)
                AAPL       IBM      MSFT      GOOG
Date
2020-11-11  0.030353 -0.006022  0.026255  0.007079
2020-11-12 -0.002343 -0.023038 -0.005126 -0.001637
2020-11-13  0.000419  0.020524  0.004967  0.015533
```

Correlation and Covariance

- The **cov** method of Series computes the **covariance** of the overlapping, non-NA, aligned-by-index values in two Series.
- Relatedly, **corr** computes the **correlation**.
- DataFrame's **corr** and **cov** methods return a full correlation or covariance **matrix** as a DataFrame, respectively.

```
returns['MSFT'].cov(returns['IBM'])
0.00015978270187216511

returns.MSFT.corr(returns.IBM)
0.5616951231541795

returns.corr()
          AAPL       IBM      MSFT      GOOG
AAPL  1.000000  0.479430  0.718757  0.656523
IBM   0.479430  1.000000  0.561695  0.522796
MSFT  0.718757  0.561695  1.000000  0.780058
GOOG  0.656523  0.522796  0.780058  1.000000
```

Unique Values, Value Counts, and Membership

- pandas has methods that extract information about the values contained in a Series.
  - Array of **unique** values
  - The **count** of each value

```
obj = pd.Series(['c', 'a', 'd',
                 'a', 'a', 'b', 'b',
                 'c', 'c'])
obj.unique()
array(['c', 'a', 'd', 'b'],
      dtype=object)


obj.value_counts()
c   3
a   3
b   2
d   1
```

Unique Values, Value Counts, and Membership

- pandas has methods that extract information about the values contained in a Series.
  - **membership** check
  - The resulting mask can be used to access matches.

```
obj = pd.Series(['c', 'a', 'd',
                 'a', 'a', 'b', 'b',
                 'c', 'c'])
mask = obj.isin(['b', 'c'])
mask
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
```

```
obj[mask]
0   c
5   b
6   b
7   c
8   c
```

Homework

- Solve the homework on **pandas**

Summary

5.1 Introduction to pandas Data Structures

5.2 Essential Functionality

5.3 Summarizing and Computing Descriptive Statistics