Object Oriented Programming (Using Python)

UNIT- III

Python Libraries:

- Pandas
  - Selection
  - Missing Data
  - Operations

Prof. R. MADANA MOHANA
Professor, Artificial Intelligence & Data Science
http://rmadanamohana.com/

Quick overview of pandas?

- https://pandas.pydata.org/docs/user_guide/10min.html#min

Selection

Getting

- Selecting a single column, which yields a Series, equivalent to df.A:

```
>>> import numpy as np
>>> import pandas as pd
>>> dates = pd.date_range("20230616", periods=6)
>>> dates
DatetimeIndex(['2023-06-16', '2023-06-17', '2023-06-
18', '2023-06-19','2023-06-20', '2023-06-21'],
              dtype='datetime64[ns]', freq='D')
```

Selection

Getting

- Selecting a single column, which yields a Series, equivalent to df.A:

```
>>>   df   =   pd.DataFrame(np.random.randn(6,   4),
index=dates, columns=list("ABCD"))
>>> df
                   A         B         C         D
2023-06-16  0.624990  0.280439 -1.135774 -1.313608
2023-06-17 -2.193261 -0.791604  0.190538 -1.416561
2023-06-18  0.490894  0.266283  0.420222  1.509806
2023-06-19  0.835607  0.468082 -0.636188  0.711356
2023-06-20  0.572577 -0.170767 -1.007022 -1.370098
2023-06-21 -0.829668  1.107512  0.742606 -1.473094
```

Selection

Getting

- Selecting a single column, which yields a Series, equivalent to df.A:

```
>>> df["A"]
2023-06-16    0.624990
2023-06-17   -2.193261
2023-06-18    0.490894
2023-06-19    0.835607
2023-06-20    0.572577
2023-06-21   -0.829668
Freq: D, Name: A, dtype: float64
>>> df.A
```

Selection

Getting

- Selecting via [] (__getitem__), which slices the rows:

```
>>> df[0:3]
                   A         B         C         D
2023-06-16  0.624990  0.280439 -1.135774 -1.313608
2023-06-17 -2.193261 -0.791604  0.190538 -1.416561
2023-06-18  0.490894  0.266283  0.420222  1.509806
>>> df["20230617":"20230620"]
                   A         B         C         D
2023-06-17 -2.193261 -0.791604  0.190538 -1.416561
2023-06-18  0.490894  0.266283  0.420222  1.509806
2023-06-19  0.835607  0.468082 -0.636188  0.711356
2023-06-20  0.572577 -0.170767 -1.007022 -1.370098
```

Selection

Selection by label

- Selection by Label using DataFrame.loc() or DataFrame.at()

DataFrame.loc()

- Access a group of rows and columns by label(s) or a boolean array.
- .loc[] is primarily label based, but may also be used with a boolean array.
- Allowed inputs are:
  - A single label, e.g. 5 or 'a', (note that 5 is interpreted as a label of the index, and never as an integer position along the index).
  - A list or array of labels, e.g. ['a', 'b', 'c'].
  - A slice object with labels, e.g. 'a':'f'.
  - A boolean array of the same length as the axis being sliced, e.g. [True, False, True].
  - An alignable boolean Series. The index of the key will be aligned before masking.
  - An alignable Index. The Index of the returned selection will be the input.
  - A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing.

Selection

Selection by label

- Selection by Label using DataFrame.loc() or DataFrame.at()

DataFrame.at()

- Access a single value for a row/column label pair.
- Similar to loc, in that both provide label-based lookups.
- Use at if you only need to get or set a single value in a DataFrame or Series.

Selection

Selection by label

- For getting a cross section using a label:

```
>>> df.loc[dates[0]]
A    0.624990
B    0.280439
C   -1.135774
D   -1.313608
Name: 2023-06-16 00:00:00, dtype: float64
```

Selection

Selection by label

- Selecting on a multi-axis by label:

```
>>> df.loc[:, ["A", "B"]]
                   A         B
2023-06-16  0.624990  0.280439
2023-06-17 -2.193261 -0.791604
2023-06-18  0.490894  0.266283
2023-06-19  0.835607  0.468082
2023-06-20  0.572577 -0.170767
2023-06-21 -0.829668  1.107512
```

Selection

Selection by label

- Showing label slicing, both endpoints are included:

```
>>> df.loc["20230616":"20230619", ["A", "B"]]
                   A         B
2023-06-16  0.624990  0.280439
2023-06-17 -2.193261 -0.791604
2023-06-18  0.490894  0.266283
2023-06-19  0.835607  0.468082
```

Selection

Selection by label

- Reduction in the dimensions of the returned object:

```
>>> df.loc["20230616", ["A", "B"]]
A    0.624990
B    0.280439
Name: 2023-06-16 00:00:00, dtype: float64
```

Selection

Selection by label

- For getting a scalar value:

```
>>> df.loc[dates[0], "A"]
0.6249895880761244
```

Selection

Selection by label

- For getting fast access to a scalar (equivalent to the prior method):

```
>>> df.at[dates[0], "A"]
0.6249895880761244
```

Selection

Selection by position

- Selection by Position using DataFrame.iloc() or DataFrame.iat().

DataFrame.iloc()

- Purely integer-location based indexing for selection by position.
- .iloc[] is primarily integer position based (from 0 to length-1 of the axis), but may also be used with a boolean array.
- Allowed inputs are:
    - An integer, e.g. 5.
    - A list or array of integers, e.g. [4, 3, 0].
    - A slice object with integers, e.g. 1:7.
    - A boolean array.
    - A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above).
    - A tuple of row and column indexes. The tuple elements consist of one of the above inputs, e.g. (0, 1).

Selection

Selection by position

- Selection by Position using DataFrame.iloc() or DataFrame.iat().

DataFrame.iat()

- Access a single value for a row/column pair by integer position.
- Similar to iloc, in that both provide integer-based lookups.
- Use iat if you only need to get or set a single value in a DataFrame or Series.

Selection

Selection by position

```
>>> import numpy as np
>>> import pandas as pd
>>> dates = pd.date_range("20230616", periods=6)
>>> dates
DatetimeIndex(['2023-06-16', '2023-06-17', '2023-06-
18', '2023-06-19','2023-06-20', '2023-06-21'],
              dtype='datetime64[ns]', freq='D')
```

Selection

Selection by position

```
>>>    df    =    pd.DataFrame(np.random.randn(6,    4),
index=dates, columns=list("ABCD"))
>>> df
                   A         B         C         D
2023-06-16  0.624990  0.280439 -1.135774 -1.313608
2023-06-17 -2.193261 -0.791604  0.190538 -1.416561
2023-06-18  0.490894  0.266283  0.420222  1.509806
2023-06-19  0.835607  0.468082 -0.636188  0.711356
2023-06-20  0.572577 -0.170767 -1.007022 -1.370098
2023-06-21 -0.829668  1.107512  0.742606 -1.473094
```

Selection

Selection by position

Select via the position of the passed integers:

```
>>> df.iloc[3]
A    1.164038
B    0.358708
C   -1.164755
D    0.136925
Name: 2023-06-19 00:00:00, dtype: float64
```

Selection

Selection by position

By integer slices, acting similar to NumPy/Python:

```
>>> df.iloc[3:5, 0:2]
                   A         B
2023-06-19  1.164038  0.358708
2023-06-20  0.802864 -0.979016
```

Selection

Selection by position

By lists of integer position locations, similar to the NumPy/Python style:

```
>>> df.iloc[[1, 2, 4], [0, 2]]
                   A         C
2023-06-17  1.429741 -0.167137
2023-06-18 -0.308334 -0.111697
2023-06-20  0.802864  0.252669
```

Selection

Selection by position

For slicing rows explicitly:

```
>>> df.iloc[1:3, :]
                   A         B         C         D
2023-06-17  1.429741 -1.425115 -0.167137  0.344657
2023-06-18 -0.308334  1.187900 -0.111697  1.718216
```

Selection

Selection by position

For slicing columns explicitly:

```
>>> df.iloc[:, 1:3]
                   B         C
2023-06-16 -1.540570  1.397791
2023-06-17 -1.425115 -0.167137
2023-06-18  1.187900 -0.111697
2023-06-19  0.358708 -1.164755
2023-06-20 -0.979016  0.252669
2023-06-21 -1.504834  0.096893
```

Selection

Selection by position

For getting a value explicitly:

```
>>> df.iloc[1, 1]
-1.4251154212429558
```

Selection

Selection by position

For getting fast access to a scalar (equivalent to the prior method):

```
>>> df.iat[1, 1]
-1.4251154212429558
```

Selection

Boolean indexing

Using a single column's values to select data:

```
>>> df[df["A"] > 0]
                   A         B         C         D
2023-06-16  0.273229 -1.540570  1.397791 -2.013290
2023-06-17  1.429741 -1.425115 -0.167137  0.344657
2023-06-19  1.164038  0.358708 -1.164755  0.136925
2023-06-20  0.802864 -0.979016  0.252669 -0.110771
2023-06-21  1.002757 -1.504834  0.096893  0.086878
```

Selection

Boolean indexing

Selecting values from a DataFrame where a boolean condition is met:

```
>>> df[df > 0]
                   A         B         C         D
2023-06-16  0.273229       NaN  1.397791       NaN
2023-06-17  1.429741       NaN       NaN  0.344657
2023-06-18       NaN  1.187900       NaN  1.718216
2023-06-19  1.164038  0.358708       NaN  0.136925
2023-06-20  0.802864       NaN  0.252669       NaN
2023-06-21  1.002757       NaN  0.096893  0.086878
```

Selection

Boolean indexing

Using the isin() method for filtering:

```
>>> df2 = df.copy()
>>>  df2["E"]  =  ["one",  "one",  "two",  "three",
"four", "three"]
```

	A	B	C	D	E
2023-06-16	0.273229	-1.540570	1.397791	-2.013290	one
2023-06-17	1.429741	-1.425115	-0.167137	0.344657	one
2023-06-18	-0.308334	1.187900	-0.111697	1.718216	two
2023-06-19	1.164038	0.358708	-1.164755	0.136925	three
2023-06-20	0.802864	-0.979016	0.252669	-0.110771	four
2023-06-21	1.002757	-1.504834	0.096893	0.086878	three

Selection

Boolean indexing

Using the isin() method for filtering:

```
>>> df2[df2["E"].isin(["two", "four"])]
                   A         B         C         D      E
2023-06-18 -0.308334  1.187900 -0.111697  1.718216    two
2023-06-20  0.802864 -0.979016  0.252669 -0.110771   four
```

Selection

Setting

Setting a new column automatically aligns the data by the indexes:

```
>>> s1 = pd.Series([1, 2, 3, 4, 5, 6], index=pd.date_range("20230617", periods=6))
>>> s1
2023-06-17    1
2023-06-18    2
2023-06-19    3
2023-06-20    4
2023-06-21    5
2023-06-22    6
Freq: D, dtype: int64
```

Selection

Setting

Setting a new column automatically aligns the data by the indexes:

```
>>> df["F"] = s1
>>> df
                   A         B         C         D         F
2023-06-16  0.273229 -1.540570  1.397791 -2.013290       NaN
2023-06-17  1.429741 -1.425115 -0.167137  0.344657  1.0
2023-06-18 -0.308334  1.187900 -0.111697  1.718216  2.0
2023-06-19  1.164038  0.358708 -1.164755  0.136925  3.0
2023-06-20  0.802864 -0.979016  0.252669 -0.110771  4.0
2023-06-21  1.002757 -1.504834  0.096893  0.086878  5.0
```

Selection

Setting

Setting values by label:

```
>>> df.at[dates[0], "A"] = 0
>>> df
                   A         B         C         D         F
2023-06-16  0.000000 -1.540570  1.397791 -2.013290       NaN
2023-06-17  1.429741 -1.425115 -0.167137  0.344657  1.000000
2023-06-18 -0.308334  1.187900 -0.111697  1.718216  2.000000
2023-06-19  1.164038  0.358708 -1.164755  0.136925  3.000000
2023-06-20  0.802864 -0.979016  0.252669 -0.110771  4.000000
2023-06-21  1.002757 -1.504834  0.096893  0.086878  5.000000
```

Selection

Setting

Setting values by position:

```
>>> df.iat[0, 1] = 0
>>> df
                   A         B         C         D         F
2023-06-16  0.000000  0.000000  1.397791 -2.013290       NaN
2023-06-17  1.429741 -1.425115 -0.167137  0.344657  1.000000
2023-06-18 -0.308334  1.187900 -0.111697  1.718216  2.000000
2023-06-19  1.164038  0.358708 -1.164755  0.136925  3.000000
2023-06-20  0.802864 -0.979016  0.252669 -0.110771  4.000000
2023-06-21  1.002757 -1.504834  0.096893  0.086878  5.000000
```

Selection

Setting

Setting by assigning with a NumPy array:

```
>>> df.loc[:, "D"] = np.array([5] * len(df))
>>> df
                   A         B         C    D    F
2023-06-16  0.000000  0.000000  1.397791  5.0  NaN
2023-06-17  1.429741 -1.425115 -0.167137  5.0  1.0
2023-06-18 -0.308334  1.187900 -0.111697  5.0  2.0
2023-06-19  1.164038  0.358708 -1.164755  5.0  3.0
2023-06-20  0.802864 -0.979016  0.252669  5.0  4.0
2023-06-21  1.002757 -1.504834  0.096893  5.0  5.0
```

Selection

Setting

Setting by assigning with a NumPy array:

```
>>> df.loc[:, "D"] = np.array([6] * len(df))
>>> df
                   A         B         C    D    F
2023-06-16  0.000000  0.000000  1.397791  6.0  NaN
2023-06-17  1.429741 -1.425115 -0.167137  6.0  1.0
2023-06-18 -0.308334  1.187900 -0.111697  6.0  2.0
2023-06-19  1.164038  0.358708 -1.164755  6.0  3.0
2023-06-20  0.802864 -0.979016  0.252669  6.0  4.0
2023-06-21  1.002757 -1.504834  0.096893  6.0  5.0
```

Selection

Setting

A where operation with setting:

```
>>> df2 = df.copy()
>>> df2[df2 > 0] = -df2
>>> df2
```

	A	B	C	D	F
2023-06-16	0.000000	0.000000	-1.397791	-6.0	NaN
2023-06-17	-1.429741	-1.425115	-0.167137	-6.0	-1.0
2023-06-18	-0.308334	-1.187900	-0.111697	-6.0	-2.0
2023-06-19	-1.164038	-0.358708	-1.164755	-6.0	-3.0
2023-06-20	-0.802864	-0.979016	-0.252669	-6.0	-4.0
2023-06-21	-1.002757	-1.504834	-0.096893	-6.0	-5.0

Missing Data

- pandas primarily uses the value `np.nan` to represent missing data.
- It is by default not included in computations.
- Reindexing allows you to change/add/delete the index on a specified axis. This returns a copy of the data:

```
>>> df1 = df.reindex(index=dates[0:4], columns=list(df.columns) + ["E"])
>>> df1.loc[dates[0] : dates[1], "E"] = 1
>>> df1
                   A         B         C    D    F    E
2023-06-16  0.000000  0.000000  1.397791  6.0  NaN  1.0
2023-06-17  1.429741 -1.425115 -0.167137  6.0  1.0  1.0
2023-06-18 -0.308334  1.187900 -0.111697  6.0  2.0  NaN
2023-06-19  1.164038  0.358708 -1.164755  6.0  3.0  NaN
```

Missing Data

- DataFrame.dropna() drops any rows that have missing data:

```
>>> df1.dropna(how="any")
>>> df1
                   A         B         C    D    F    E
2023-06-17  1.429741 -1.425115 -0.167137  6.0  1.0  1.0
```

Missing Data

- DataFrame.fillna() fills missing data:

```
>>> df1.fillna(value=5)
>>> df1
                   A         B         C    D    F    E
2023-06-16  0.000000  0.000000  1.397791  6.0  5.0  1.0
2023-06-17  1.429741 -1.425115 -0.167137  6.0  1.0  1.0
2023-06-18 -0.308334  1.187900 -0.111697  6.0  2.0  5.0
2023-06-19  1.164038  0.358708 -1.164755  6.0  3.0  5.0
```

Missing Data

- isna() gets the boolean mask where values are nan:

```
>>> pd.isna(df1)
>>> df1
                A      B      C      D      F      E
2023-06-16  False  False  False  False   True  False
2023-06-17  False  False  False  False  False  False
2023-06-18  False  False  False  False  False   True
2023-06-19  False  False  False  False  False   True
```

Operations

Stats

- Operations in general *exclude* missing data.
- Performing a descriptive statistic:

```
>>> df
                   A         B         C    D    F
2023-06-16  0.000000  0.000000  1.397791  6.0  NaN
2023-06-17  1.429741 -1.425115 -0.167137  6.0  1.0
2023-06-18 -0.308334  1.187900 -0.111697  6.0  2.0
2023-06-19  1.164038  0.358708 -1.164755  6.0  3.0
2023-06-20  0.802864 -0.979016  0.252669  6.0  4.0
2023-06-21  1.002757 -1.504834  0.096893  6.0  5.0
```

Operations

Stats

- Performing a descriptive statistic:

```
>>> df.mean() #column wise mean
A    0.681844
B   -0.393726
C    0.050627
D    6.000000
F    3.000000
dtype: float64
```

Operations

Stats

- Same operation on the other axis:

```
>>> df.mean(1)#row wise mean
2023-06-16    1.849448
2023-06-17    1.367498
2023-06-18    1.753574
2023-06-19    1.871598
2023-06-20    2.015303
2023-06-21    2.118963
Freq: D, dtype: float64
```

Operations

Stats

- Operating with objects that have different dimensionality and need alignment.
- In addition, pandas automatically broadcasts along the specified dimension:

```
>>> s = pd.Series([1, 3, 5, np.nan, 6, 8], index=dates).shift(2)
>>> s
2023-06-16    NaN
2023-06-17    NaN
2023-06-18    1.0
2023-06-19    3.0
2023-06-20    5.0
2023-06-21    NaN
Freq: D, dtype: float64
```

Operations

Stats

- Operating with objects that have different dimensionality and need alignment.
- In addition, pandas automatically broadcasts along the specified dimension:

```
>>> df.sub(s, axis="index")
                   A         B         C    D    F
2023-06-16       NaN       NaN       NaN  NaN  NaN
2023-06-17       NaN       NaN       NaN  NaN  NaN
2023-06-18 -1.308334  0.187900 -1.111697  5.0  1.0
2023-06-19 -1.835962 -2.641292 -4.164755  3.0  0.0
2023-06-20 -4.197136 -5.979016 -4.747331  1.0 -1.0
2023-06-21       NaN       NaN       NaN  NaN  NaN
```

Operations

Apply

- DataFrame.apply() applies a user defined function to the data:

```
>>> df
                   A         B         C    D    F
2023-06-16  0.000000  0.000000  1.397791  6.0  NaN
2023-06-17  1.429741 -1.425115 -0.167137  6.0  1.0
2023-06-18 -0.308334  1.187900 -0.111697  6.0  2.0
2023-06-19  1.164038  0.358708 -1.164755  6.0  3.0
2023-06-20  0.802864 -0.979016  0.252669  6.0  4.0
2023-06-21  1.002757 -1.504834  0.096893  6.0  5.0
```

Operations

Apply

- DataFrame.apply() applies a user defined function to the data:

```
>>> df.apply(np.cumsum) # cumulative sum column wise
                   A         B         C     D     F
2023-06-16  0.000000  0.000000  1.397791   6.0   NaN
2023-06-17  1.429741 -1.425115  1.230654  12.0   1.0
2023-06-18  1.121407 -0.237215  1.118957  18.0   3.0
2023-06-19  2.285445  0.121493 -0.045798  24.0   6.0
2023-06-20  3.088308 -0.857523  0.206872  30.0  10.0
2023-06-21  4.091065 -2.362357  0.303764  36.0  15.0
```

Operations

Apply

- DataFrame.apply() applies a user defined function to the data:

```
>>> df.apply(lambda x: x.max() - x.min()) # column wise
A    1.738075
B    2.692734
C    2.562546
D    0.000000
F    4.000000
dtype: float64
```

Operations

String Methods

Series is equipped with a set of string processing methods in the str attribute that make it easy to operate on each element of the array, as in the code snippet below.

Note that pattern-matching in str generally uses regular expressions by default (and in some cases always uses them).

Operations

String Methods

```
>>> s = pd.Series(["A", "B", "C", "Aaba", "Baca", np.nan, "CABA", "dog", "cat"])
>>> s.str.lower()
0       a
1       b
2       c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object
```