Data Science Workshop-1

ITER, SOA University

Centre for Data Science
SOA University

pandas

Introduction

Imagine you're an explorer in a vast digital world, where every piece of information you need is spread across countless islands. Some of these islands are small spreadsheets, others are massive databases, and some even contain messy, tangled webs of data. But you, the brave data explorer, have a secret tool: a magical panda.

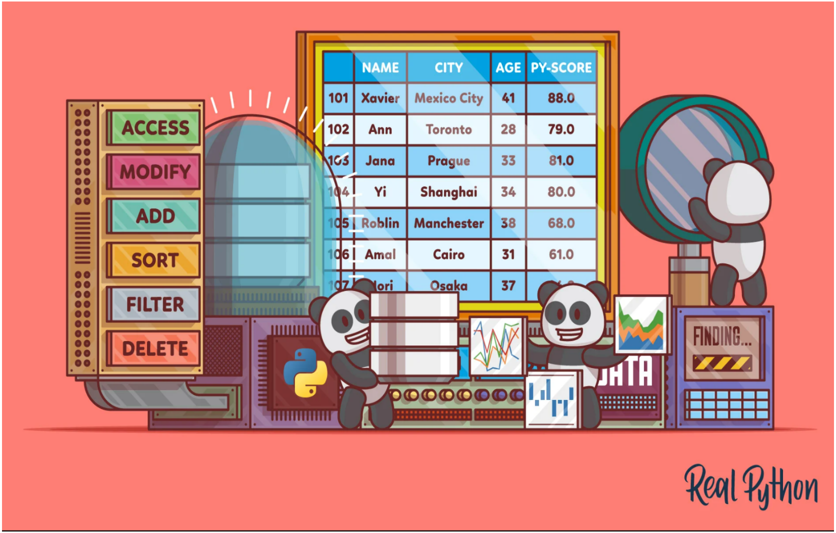
Your Secret Tool

This panda is no ordinary bear; it has special powers that can help you wrangle data, make sense of large tables, and turn confusing information into clear, valuable insights. With just a little guidance, this magical panda will help you explore, clean, and analyze the data landscapes you encounter.

pandas

- pandas derived from 'panel data'. In econometrics this is a term for 'tabular data'
- The biggest difference is that pandas is designed for working with tabular or heterogeneous data.
- Basic data structures in pandas
  - series: One-dimensional array-like object containing a sequence of values (of similar types to NumPy types) of the same type and an associated array of data labels, called its index.
  - DataFrame: a two-dimensional data structure that holds data like a two-dimension array or a table with rows and columns.

Contents in Pandas



Contents

- Series
- Dictionary to pandas series and data frame
- Selecting rows and columns
- Re-indexing, dropping entries, filter
- Arithmetic and data alignment
- Summarizing and computing descriptive statistics
- Values counts, covariance and correlation

Series

```
import pandas as pd
```

```
x=pd.Series([1,2,3])
x
```

```
0    1
1    2
2    3
dtype: int64
```

```
x.array
```

```
<PandasArray>
[1, 2, 3]
Length: 3, dtype: int64
```

```
x.index
```

```
RangeIndex(start=0, stop=3, step=1)
```

```
y = pd.Series([4, 7, -5], index=["x",'y','z'])
y
```

```
x    4
y    7
z   -5
dtype: int64
```

Basics of indexing

- The index on the left and the values on the right.
- If we will not specify an index for the data, a default one consisting of the integers 0 through N - 1 (where N is the length of the data) is created.
- You can get the array representation and index object of the Series via its array and index attributes
- want to create a Series with an index identifying each data point with a label

Selecting and filtering

- Like numpy arrays we can use labels in the index when selecting single values or a set of values.
- NumPy-like operations, such as filtering with a Boolean array, scalar multiplication, or applying math functions, will preserve the index-value link

```
: y['x']

: 4

: y[['x','z']]

: x     4
  z    -5
  dtype: int64

: y[y > 0]

: x     4
  y     7
  dtype: int64

: y*2

: x      8
  y     14
  z    -10
```

Dictionary to pandas series

- Series can be thought as a fixed-length, ordered dictionary, as it is a mapping of index values to data values.
- If we have data contained in a Python dictionary, you can create a Series from it by passing the dictionary to the series method
- A Series can be converted back to a dictionary with its to_dict method

```
import pandas as pd
```

```
sdata = {"Adishree": 2241020023, "Sushree": 2241019629,
         "Anuska": 2241019579, "Omkar": 2241019335,
         'Goswami':2241019331}
S = pd.Series(sdata)
S
```

```
Adishree     2241020023
Sushree      2241019629
Anuska       2241019579
Omkar        2241019335
Goswami      2241019331
dtype: int64
```

```
S.to_dict()
```

```
{'Adishree': 2241020023,
 'Sushree': 2241019629,
 'Anuska': 2241019579,
 'Omkar': 2241019335,
 'Goswami': 2241019331}
```

Dictionary to pandas series

- when you are only passing a dictionary, the index in the resulting Series will respect the order of the keys
- if we want to see them in a different order, that can be done by passing a list containing the dictionary keys in the desired order to the index keyword argument

```
L=['Rishita','Goswami','Sweta','Omkar','Anuska' ]
M=pd.Series(sdata,index=L)
M
```

```
Rishita              NaN
Goswami     2.241019e+09
Sweta                NaN
Omkar       2.241019e+09
Anuska      2.241020e+09
dtype: float64
```

Missing values in pandas

- As no registration number for "Some students" was found, it appears as NaN (Not a Number), which is considered in pandas to mark missing or NA values
- `isna` and `notna` functions in pandas should be used to detect missing data

```
pd.isna(M)
```

```
Rishita      True
Goswami     False
Sweta        True
Omkar       False
Anuska      False
dtype: bool
```

```
pd.notna(M)
```

```
Rishita     False
Goswami      True
Sweta       False
Omkar        True
Anuska       True
dtype: bool
```

Missing values in pandas

- Automatically aligns by index label in arithmetic operations:

```
data1={'Odisha':17,'Bengal':15}
data2={'Odisha':4,'Delhi':7}
d1=pd.Series(data1)
d2=pd.Series(data2)
d1+d2
```

```
Bengal      NaN
Delhi       NaN
Odisha     21.0
dtype: float64
```

Data frame creation

- A DataFrame represents a rectangular table of data and contains an ordered, named collection of columns, each of which can be a different data type
- The DataFrame has both a row and column index; it can be thought of as a dictionary of Series all sharing the same index

[1]:
```
import pandas as pd
data={"State":["Delhi (NCT)","Gujarat","Maharashtra","Odisha","Uttarakhand"],
     "CM":["Atishi Marlena","Bhupendra Patel","Eknath Shinde","Mohan Charan Majhi","Pushkar Singh Dhami"],
     "Pop":[4.37,10.43,13.10,5.2,3.2]}
frame=pd.DataFrame(data)
frame
```

[1]:

	State	CM	Pop
0	Delhi (NCT)	Atishi Marlena	4.37
1	Gujarat	Bhupendra Patel	10.43
2	Maharashtra	Eknath Shinde	13.10
3	Odisha	Mohan Charan Majhi	5.20
4	Uttarakhand	Pushkar Singh Dhami	3.20

head and tail method

- The head method selects only the first five rows
- The tail returns the last five rows.

[3]:
```
frame.head()
```

[3]:

	State	CM	Pop
0	Delhi (NCT)	Atishi Marlena	4.37
1	Gujarat	Bhupendra Patel	10.43
2	Maharashtra	Eknath Shinde	13.10
3	Odisha	Mohan Charan Majhi	5.20
4	Uttarakhand	Pushkar Singh Dhami	3.20

[5]:
```
frame.tail()
```

[5]:

	State	CM	Pop
0	Delhi (NCT)	Atishi Marlena	4.37
1	Gujarat	Bhupendra Patel	10.43
2	Maharashtra	Eknath Shinde	13.10
3	Odisha	Mohan Charan Majhi	5.20
4	Uttarakhand	Pushkar Singh Dhami	3.20

Naming the columns

- If you specify a sequence of columns, the DataFrame's columns will be arranged in that order
- If you pass a column that isn't contained in the dictionary, it will appear with missing values in the result
- data.columns gives us all the columns in the data

[7]:
```
frame1=pd.DataFrame(data,columns=["CM","Pop","State","Loan_amt"])
frame1
```

[7]:

	CM	Pop	State	Loan_amt
0	Atishi Marlena	4.37	Delhi (NCT)	NaN
1	Bhupendra Patel	10.43	Gujarat	NaN
2	Eknath Shinde	13.10	Maharashtra	NaN
3	Mohan Charan Majhi	5.20	Odisha	NaN
4	Pushkar Singh Dhami	3.20	Uttarakhand	NaN

[9]:
```
frame1.columns
```

[9]:
```
Index(['CM', 'Pop', 'State', 'Loan_amt'], dtype='object')
```

Retrieving a column

- To retrieve a column, either use dictionary like syntax or column name as a attribute

```
[11]: frame1['State']

[11]: 0    Delhi (NCT)
      1        Gujarat
      2    Maharashtra
      3         Odisha
      4    Uttarakhand
      Name: State, dtype: object
```

```
[13]: frame1.CM

[13]: 0          Atishi Marlena
      1         Bhupendra Patel
      2           Eknath Shinde
      3       Mohan Charan Majhi
      4      Pushkar Singh Dhami
      Name: CM, dtype: object
```

Updating a column

- Columns can be modified by assignment.
- The empty Governor column could be assigned a scalar value or an array of values

[17]:
```
frame1['Loan_amt']=23
frame1
```

[17]:

	CM	Pop	State	Loan_amt
0	Atishi Marlena	4.37	Delhi (NCT)	23
1	Bhupendra Patel	10.43	Gujarat	23
2	Eknath Shinde	13.10	Maharashtra	23
3	Mohan Charan Majhi	5.20	Odisha	23
4	Pushkar Singh Dhami	3.20	Uttarakhand	23

[19]:
```
frame1['Loan_amt']=[3.4,7.8,6.7,9.8,5.3]
frame1
```

[19]:

	CM	Pop	State	Loan_amt
0	Atishi Marlena	4.37	Delhi (NCT)	3.4
1	Bhupendra Patel	10.43	Gujarat	7.8
2	Eknath Shinde	13.10	Maharashtra	6.7
3	Mohan Charan Majhi	5.20	Odisha	9.8
4	Pushkar Singh Dhami	3.20	Uttarakhand	5.3

Updating a data frame

- When you are assigning lists or arrays to a column, the value's length must match the length of the DataFrame
- If you assign a Series, its labels will be realigned exactly to the data frames index, inserting missing values in any index values not present

[31]:
```
frame2=pd.DataFrame(data,columns=["CM","Pop","State","Loan_amt"])
val=pd.Series([3.4,6.7,9.8],index=[0,1,3])
frame2['Loan_amt']=val
frame2
```

[31]:

	CM	Pop	State	Loan_amt
0	Atishi Marlena	4.37	Delhi (NCT)	3.4
1	Bhupendra Patel	10.43	Gujarat	6.7
2	Eknath Shinde	13.10	Maharashtra	NaN
3	Mohan Charan Majhi	5.20	Odisha	9.8
4	Pushkar Singh Dhami	3.20	Uttarakhand	NaN

creating new col and deleting

- Assigning a column that doesn't exist will create a new column.
- The del method can then be used to remove this column

```
[45]: frame1['Capital']='Bhubaneswar'
      frame1
```

[45]:

	CM	Pop	State	Loan_amt	Capital
0	Atishi Marlena	4.37	Delhi (NCT)	3.4	Bhubaneswar
1	Bhupendra Patel	10.43	Gujarat	7.8	Bhubaneswar
2	Eknath Shinde	13.10	Maharashtra	6.7	Bhubaneswar
3	Mohan Charan Majhi	5.20	Odisha	9.8	Bhubaneswar
4	Pushkar Singh Dhami	3.20	Uttarakhand	5.3	Bhubaneswar

```
[47]: del frame1['Capital']
```

```
[49]: frame1
```

[49]:

	CM	Pop	State	Loan_amt
0	Atishi Marlena	4.37	Delhi (NCT)	3.4
1	Bhupendra Patel	10.43	Gujarat	7.8
2	Eknath Shinde	13.10	Maharashtra	6.7
3	Mohan Charan Majhi	5.20	Odisha	9.8
4	Pushkar Singh Dhami	3.20	Uttarakhand	5.3

Nested Dictionaries

- If a 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

```
GDP = {"India": {2015: 2.1, 2020: 2.7, 2025: 5},
       "USA": {2015: 18.4, 2020: 21.06,2025:25},
      "Japan": {2015: 4.4, 2020: 5.0, 2025: 5.4}}
frame3 = pd.DataFrame(GDP)
frame3
```

	India	USA	Japan
2015	2.1	18.40	4.4
2020	2.7	21.06	5.0
2025	5.0	25.00	5.4

Other ways of creating DataFrame

- Creating DataFrame from dict of lists , dict of dicts
- From 2-dim numpy arrays

```
#not more than 2-d
a=np.array([[1,2],[3,4],[5,6]])
a=pd.DataFrame(a,index=['r1','r2','r3'],columns=['c1','c2'])
#a.columns=['c1','c2']
#a.index=['r1','r2','r3']
a
```

	c1	c2
r1	1	2
r2	3	4
r3	5	6

Other ways of creating DataFrame

- Creating Pandas DataFrame from lists of lists.

```
data = [['Virat', 35], ['Dhoni', 42], ['Shubman', 24]]
df = pd.DataFrame(data, columns=['Name', 'Age'])
df
```

	Name	Age
0	Virat	35
1	Dhoni	42
2	Shubman	24

Other ways of creating DataFrame

- Creating Dataframe from list of dicts

```
data = [{'a': 1, 'b': 2, 'c': 3},
        {'a': 10, 'b': 20, 'c': 30,'d':7}]

df = pd.DataFrame(data)
df
```

	a	b	c	d
0	1	2	3	NaN
1	10	20	30	7.0

Other ways of creating DataFrame

- Creating DataFrame from Dictionary of series

```
d = {'one': pd.Series([10, 20, 30, 40],
                      index=['a', 'b', 'c', 'd']),
     'two': pd.Series([10, 20, 30, 40],
                      index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(d)
df
```

	one	two
a	10	10
b	20	20
c	30	30
d	40	40

Naming index and column

- We can use name attribute for naming the index and columns

```
import pandas as pd
GDP = {"India": {2020: 3.8, 2025: 5},
      "USA": {2020: 13.5, 2025: 15}}
frame3 = pd.DataFrame(GDP)
frame3.index.name='Year'
frame3.columns.name='Country'
frame3
```

Country	India	USA
Year		
2020	3.8	13.5
2025	5.0	15.0

Index objects in pandas

- Index is an immutable sequence (ordered multiset) used for indexing DataFrame and Series
- standard python indexing and slicing notation works for index objects
- create an index object using pd.Index() method
- NumPy attributes like shape, size, ndim and dtype works for index objects
- Immutable is the key difference between numpy array and indices
- many set operations like union, intersection, symmetric_difference can be used for index

Index objects in pandas

```
import pandas as pd
df=pd.DataFrame(['a','b','c'],index=[1,2,3])
I=df.index
print(I[2])
```

```
3
```

```
labels = pd.Index(np.arange(1,4))
print('labels',labels)
pd.Series([1.5, -2.5, 0], index=labels)
```

```
Int64Index([1, 2, 3], dtype='int64')

1     1.5
2    -2.5
3     0.0
dtype: float64
```

```
print(I.shape)
print(I.size)
print(I.dtype)
print(I.ndim)
```

```
(3,)
3
int64
1
```

```
I1=pd.Index([1,2,3,4,5,6])
I2=pd.Index([3,4,5,6,7,8])
print(I1.union(I2))
print(I1.intersection(I2))
print(I1.symmetric_difference(I2))
print(I1.append(I2))
```

```
Int64Index([1, 2, 3, 4, 5, 6, 7, 8], dtype='int64')
Int64Index([3, 4, 5, 6], dtype='int64')
Int64Index([1, 2, 7, 8], dtype='int64')
Int64Index([1, 2, 3, 4, 5, 6, 3, 4, 5, 6, 7, 8], dtype='int64')
```

Reindexing

- reindex is used to create a new object with the values rearranged
- DataFrame.reindex(labels,index,columns, axis, method, copy, level, fill_value, limit, tolerance)
- labels and index works same: New index labels

```
import pandas as pd
import numpy as np
frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
                     index=["i", "ii", "iii"],columns=["A", "B", "C"])
frame
```

	A	B	C
i	0	1	2
ii	3	4	5
iii	6	7	8

```
frame2 = frame.reindex(["i", "iii", "ii"])
frame2
```

	A	B	C
i	0	1	2
iii	6	7	8
ii	3	4	5

```
frame2 = frame.reindex(index=["i", "iii", "ii"])
frame2
```

	A	B	C
i	0	1	2
iii	6	7	8
ii	3	4	5

Reindexing

- columns: labels for the columns
- axis: Axis to target. Can be either the axis name (‘index’, ‘columns’) or number (0, 1)
- fill_value: Value to use for missing value

```
frame2 = frame.reindex(columns=["A", "B"])
frame2
```

	A	B
i	0	1
ii	3	4
iii	6	7

```
frame2 = frame.reindex(["ii","A","D", "i"],axis=0)
frame2
```

	A	B	C
ii	3.0	4.0	5.0
A	NaN	NaN	NaN
D	NaN	NaN	NaN
i	0.0	1.0	2.0

```
frame2 = frame.reindex(["C","A","D", "B"],axis=1)
frame2
```

	C	A	D	B
i	2	0	NaN	1
ii	5	3	NaN	4
iii	8	6	NaN	7

```
frame2 = frame.reindex(["ii","A","D", "i"],axis=0,fill_value=7)
frame2
```

	A	B	C
ii	3	4	5
A	7	7	7
D	7	7	7
i	0	1	2

Reindexing

- method: use for filling holes in reindexed DataFrame.
- Note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.
- None (default): don't fill gaps, ffill: Propagate last valid observation forward to next valid, bfill: Use next valid observation to fill gap

```
frame1 = pd.Series(["A", "B", "C"],
                   index=[0, 2, 4])
frame1
```

```
0    A
2    B
4    C
dtype: object
```

```
frame1.reindex(np.arange(6))
```

```
0      A
1    NaN
2      B
3    NaN
4      C
5    NaN
```

```
frame1.reindex(np.arange(6),
               method="ffill")
```

```
0    A
1    A
2    B
3    B
4    C
5    C
dtype: object
```

```
frame1.reindex(np.arange(6),
               method="bfill")
```

```
0      A
1      B
2      B
3      C
4      C
5    NaN
```

Dropping entries from an Axis

- Remove rows or columns by specifying label names and corresponding axis, or by directly specifying index or column names

```
df = pd.DataFrame(np.arange(16).reshape((4, 4)),
                  index=["Aae", "Bee", "Cee", "Dee"],
                  columns=["one", "two", "three", "four"])
df
```

	one	two	three	four
Aae	0	1	2	3
Bee	4	5	6	7
Cee	8	9	10	11
Dee	12	13	14	15

```
df.drop('Aae')
```

	one	two	three	four
Bee	4	5	6	7
Cee	8	9	10	11
Dee	12	13	14	15

```
df.drop('one')
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recer
```

```
df.drop(index=['Aae','Bee'])
```

	one	two	three	four
Cee	8	9	10	11
Dee	12	13	14	15

```
df.drop('one',axis=1)
```

	two	three	four
Aae	1	2	3
Bee	5	6	7
Cee	9	10	11
Dee	13	14	15

```
df.drop(columns=["two", "four"],index=['Aae'])
```

	one	three
Bee	4	6
Cee	8	10
Dee	12	14

Indexing, selection, filter in Series

- Series indexing works analogously to NumPy array indexing, except you can use the Series's index values instead of only integers

```
obj = pd.Series(np.arange(4.),
                index=["a", "b", "c", "d"])
obj
```

```
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64
```

```
print('obj["b"]=',obj["b"])
print('obj[1]=',obj[1])
```

```
obj["b"]= 1.0
obj[1]= 1.0
```

```
print(obj[2:4])
```

```
c    2.0
d    3.0
dtype: float64
```

```
print(obj[["b", "a", "d"]])
```

```
b    1.0
a    0.0
d    3.0
dtype: float64
```

```
print(obj[[1, 3]])
```

```
b    1.0
d    3.0
dtype: float64
```

```
print(obj[obj < 2])
```

```
a    0.0
b    1.0
dtype: float64
```

Indexing in DataFrame

- Indexing into a DataFrame retrieves one or more columns either with a single value or sequence
- it is more preferable to use loc and iloc methods

```
data=pd.DataFrame(np.arange(16).reshape((4,4)),
                  index=['r1','r2','r3','r4'],
                  columns=['c1','c2','c3','c4'])
data
```

	c1	c2	c3	c4
r1	0	1	2	3
r2	4	5	6	7
r3	8	9	10	11
r4	12	13	14	15

```
data['c1']
```

```
r1     0
r2     4
r3     8
r4    12
Name: c1, dtype: int32
```

```
data[['c1','c3']]
```

	c1	c3
r1	0	2
r2	4	6
r3	8	10
r4	12	14

Indexing in DataFrame

```
data[:2]
```

	c1	c2	c3	c4
r1	0	1	2	3
r2	4	5	6	7

```
data[data['c3']>3]
```

	c1	c2	c3	c4
r2	4	5	6	7
r3	8	9	10	11
r4	12	13	14	15

```
data>5
```

	c1	c2	c3	c4
r1	False	False	False	False
r2	False	False	True	True
r3	True	True	True	True
r4	True	True	True	True

```
data[data['c3']>3]=10
data
```

	c1	c2	c3	c4
r1	0	1	2	3
r2	10	10	10	10
r3	10	10	10	10
r4	10	10	10	10

loc and iloc in pandas

- loc and iloc for label-based and integer-based indexing
- loc[] is primarily label based, but may also be used with a boolean array.
- Single label. Note this returns the row as a Series
- List of labels. Note using [[]] returns a DataFrame

```
import pandas as pd
df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
     index=['Cobra', 'Viper', 'RatSnake'],
     columns=['max_speed', 'length'])
df
```

	max_speed	length
Cobra	1	2
Viper	4	5
RatSnake	7	8

```
df.loc['Viper']
```

```
max_speed     4
length        5
Name: Viper, dtype: int64
```

```
df.loc[['Viper','Cobra']]
```

	max_speed	length
Viper	4	5
Cobra	1	2

```
df.loc['Viper','length']
```

```
5
```

loc and iloc in pandas

- Slice with labels for row and single label for column.
- Boolean list with the same length as the row axis
- Conditional that returns a boolean Series
- Conditional that returns a boolean Series with column labels specified

```
df.loc['Cobra':'Viper', 'max_speed']
```

```
Cobra     1
Viper     4
Name: max_speed, dtype: int64
```

```
df.loc[[False, False, True]]
```

	max_speed	length
RatSnake	7	8

```
df.loc[df['length'] > 6]
```

	max_speed	length
RatSnake	7	8

```
df.loc[df['length']>6,'max_speed']
```

```
RatSnake     7
Name: max_speed, dtype: int64
```

loc and iloc in pandas

- iloc: integer-location based indexing for selection by position

```
data=pd.DataFrame(np.arange(16).reshape((4,4)),
                  index=['r1','r2','r3','r4'],
                  columns=['c1','c2','c3','c4'])
data
```

	c1	c2	c3	c4
r1	0	1	2	3
r2	4	5	6	7
r3	8	9	10	11
r4	12	13	14	15

```
data.iloc[0]
```

```
c1     0
c2     1
c3     2
c4     3
Name: r1, dtype: int32
```

```
data.iloc[:,0]
```

```
r1      0
r2      4
r3      8
r4     12
Name: c1, dtype: int32
```

loc and iloc in pandas

```
data.iloc[[1,3]]
```

	c1	c2	c3	c4
r2	4	5	6	7
r4	12	13	14	15

```
data.iloc[[True,True,False,False]]
```

	c1	c2	c3	c4
r1	0	1	2	3
r2	4	5	6	7

```
data.iloc[0:2,1:3]
```

	c2	c3
r1	1	2
r2	5	6

```
data.iloc[[True,True,False,False],
          [True,True,False,False]]
```

	c1	c2
r1	0	1
r2	4	5

```
data.at['r2','c3']
```

6

```
data.iat[1,2]
```

6

Indexing in dataframe

Table 5.4: Indexing options with DataFrame

Type	Notes
`df[column]`	Select single column or sequence of columns from the DataFrame; special case conveniences: Boolean array (filter rows), slice (slice rows), or Boolean DataFrame (set values based on some criterion)
`df.loc[rows]`	Select single row or subset of rows from the DataFrame by label
`df.loc[:, cols]`	Select single column or subset of columns by label
`df.loc[rows, cols]`	Select both row(s) and column(s) by label
`df.iloc[rows]`	Select single row or subset of rows from the DataFrame by integer position
`df.iloc[:, cols]`	Select single column or subset of columns by integer position
`df.iloc[rows, cols]`	Select both row(s) and column(s) by integer position
`df.at[row, col]`	Select a single scalar value by row and column label
`df.iat[row, col]`	Select a single scalar value by row and column position (integers)
`reindex` method	Select either rows or columns by labels

Experiement loc and iloc methods on series object

Integer indexing pitfalls

Let us consider an example

```
ser = pd.Series(np.arange(3.))
ser
```

```
0    0.0
1    1.0
2    2.0
dtype: float64
```

```
ser.iloc[-1]
```

```
2.0
```

```
ser[-1]
```

```
---------------------------------------------
ValueError                                  T
```

```
ser = pd.Series(np.arange(3.),index=['a','b','c'])
ser
```

```
a    0.0
b    1.0
c    2.0
dtype: float64
```

```
ser[-1]
```

```
2.0
```

```
ser.iloc[-1]
```

```
2.0
```

If you have an axis index containing integers, data selection will always be label oriented.

NumPy ufuncs

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

```
frame = pd.DataFrame(np.random.standard_normal((4, 3)),
                     columns=list("bde"),
                     index=["India", "USA", "Japan", "China"])

frame
```

	b	d	e
India	0.680518	0.592259	0.446467
USA	-0.478302	0.529713	-1.430172
Japan	1.246007	0.617735	-0.864200
China	-1.488501	-0.085932	2.205331

```
np.abs(frame)
```

	b	d	e
India	0.680518	0.592259	0.446467
USA	0.478302	0.529713	1.430172
Japan	1.246007	0.617735	0.864200
China	1.488501	0.085932	2.205331

Function Application and mapping

- Apply a function along an axis of the DataFrame
- either the DataFrame's index (axis=0) or the DataFrame's columns (axis=1)

```
df = pd.DataFrame([[4, 9]] * 3,
                  columns=['A', 'B'])
df
```

	A	B
0	4	9
1	4	9
2	4	9

```
df.apply(np.sum, axis=0)
```

```
A    12
B    27
dtype: int64
```

```
df.apply(np.sum, axis=1)
```

```
0    13
1    13
2    13
dtype: int64
```

```
def f1(x):
    return x.max() - x.min()
df.apply(f1, axis=1)
```

```
0    5
1    5
2    5
dtype: int64
```

```
def f1(x):
    return x.max() - x.min()
df.apply(f1, axis=0)
```

```
A    0
B    0
dtype: int64
```

Exercise

1. create a 4*4 matrix, where each row contains Student's name, registration no, sec and cgpa.
D={'Name':['A',"B","C","D"],'Reg_no':[1,2,3,4],
'Sec':['i','ii','iii','iv'],'CGPA':[7,8,9,10]} Find the name, registration no of the student with highest cgpa.(Use np.max on cgpa column) What is the row and column number of the cell with the highest CGPA? (Convert dataframe to numpy array and see which indices has maximum value)

Exercise

1. create a 4*4 matrix, where each row contains Student's name, registration no, sec and cgpa.
   D={'Name':['A',"B","C","D"],'Reg_no':[1,2,3,4], 'Sec':['i','ii','iii','iv'],'CGPA':[7,8,9,10]} Find the name, registration no of the student with highest cgpa.(Use np.max on cgpa column) What is the row and column number of the cell with the highest CGPA? (Convert dataframe to numpy array and see which indices has maximum value)
   Ans df.loc[df['CGPA']==np.max(df.CGPA),['Name','Reg_no']]
   row, col = np.where(df.to_numpy() == np.max(df.CGPA))

Exercise

1. create a 4*4 matrix, where each row contains Student's name, registration no, sec and cgpa.
   D={'Name':['A',"B","C","D"],'Reg_no':[1,2,3,4], 'Sec':['i','ii','iii','iv'],'CGPA':[7,8,9,10]} Find the name, registration no of the student with highest cgpa.(Use np.max on cgpa column) What is the row and column number of the cell with the highest CGPA? (Convert dataframe to numpy array and see which indices has maximum value)
   Ans df.loc[df['CGPA']==np.max(df.CGPA),['Name','Reg_no']]
   row, col = np.where(df.to_numpy() == np.max(df.CGPA))
2. Count the number of missing values in each column of df. Which column has the maximum number of missing values?(create a function to count nan values in a column, use df.apply and that function)

Exercise

1. create a 4*4 matrix, where each row contains Student's name, registration no, sec and cgpa.
   D={'Name':['A',"B","C","D"],'Reg_no':[1,2,3,4], 'Sec':['i','ii','iii','iv'],'CGPA':[7,8,9,10]} Find the name, registration no of the student with highest cgpa.(Use np.max on cgpa column) What is the row and column number of the cell with the highest CGPA? (Convert dataframe to numpy array and see which indices has maximum value)
   Ans df.loc[df['CGPA']==np.max(df.CGPA),['Name','Reg_no']]
   row, col = np.where(df.to_numpy() == np.max(df.CGPA))
2. Count the number of missing values in each column of df. Which column has the maximum number of missing values?(create a function to count nan values in a column, use df.apply and that function) **Ans**
   missings_each_col = df.apply(lambda x: x.isnull().sum())
   missings_each_col.argmax()

Exercise

1. Create a 4*4 matrix, entries from np.arange(16). Reverse the entries of the matrix. The matrix will look like $\begin{bmatrix} 15 & 14 & 13 & 12 \\ 11 & 10 & 9 & 8 \\ 7 & 6 & 5 & 4 \\ 3 & 2 & 1 & 0 \end{bmatrix}$

Exercise

1. Create a 4*4 matrix, entries from np.arange(16). Reverse the entries of the matrix. The matrix will look like $\begin{bmatrix} 15 & 14 & 13 & 12 \\ 11 & 10 & 9 & 8 \\ 7 & 6 & 5 & 4 \\ 3 & 2 & 1 & 0 \end{bmatrix}$

```
x=pd.DataFrame(np.arange(16).reshape((4,4)))
x.loc[::-1,::-1]
```

2. Create a 4*4 matrix where each entries are from standard normal distribution. Add a column to this data frame, which contains row wise minimums.

Exercise

1. Create a 4*4 matrix, entries from np.arange(16). Reverse the entries of the matrix. The matrix will look like $\begin{bmatrix} 15 & 14 & 13 & 12 \\ 11 & 10 & 9 & 8 \\ 7 & 6 & 5 & 4 \\ 3 & 2 & 1 & 0 \end{bmatrix}$

```
x=pd.DataFrame(np.arange(16).reshape((4,4)))
x.loc[::-1,::-1]
```

2. Create a 4*4 matrix where each entries are from standard normal distribution. Add a column to this data frame, which contains row wise minimums.

```
df=pd.DataFrame(np.random.randn(16).reshape((4,4)))
print(df)
row_max=df.apply(np.max,axis=1)
df['row_max']=row_max
df
```

Exercise

1. Create a 4*4 matrix entries from standard normal distribution. Add another row where all entries are NaN. Replace missing values in 'col1' and 'col2' columns with their respective mean.

```
import pandas as pd
import numpy as np
dff=pd.DataFrame(np.random.randn(16).reshape((4,4)),
                 index=['a','b','c','d'],
                 columns=['c1','c2','c3','c4'])
df=pd.DataFrame(dff,index=['a','b','c','d','e'])
df[['c1', 'c2']].apply(lambda x: x.fillna(x.mean()))
```

Arithmetic and Data Alignment

- Arithmetic operation between objects that have different indexes, if any index pairs are not the same, the respective index in the result will be the union of the index pairs.
- Using the add method on df1, I pass df2 and an argument to fill_value, which substitutes the passed value for any missing values in the operation

```
df1 = pd.DataFrame(np.arange(12).reshape((3, 4)),
                   columns=list("abcd"))
df1
```

	a	b	c	d
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

```
df2 = pd.DataFrame(np.arange(20).reshape((4, 5)),
                   columns=list("abcde"))
df2
```

	a	b	c	d	e
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19

```
df1.add(df2)
```

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	NaN
1	9.0	11.0	13.0	15.0	NaN
2	18.0	20.0	22.0	24.0	NaN
3	NaN	NaN	NaN	NaN	NaN

```
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	11.0	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

Arithmetic and Data Alignment

- If data in both corresponding DataFrame locations is missing the result will be missing.
- df1.add(df2) is equivalent to df2.radd(df1), r stands for reverse, the arguments are reversed
- Experiment with sub, rsub, div, rdiv, floordiv, rfloordiv, mul, rmul, pow, rpow

```
df2.loc[1, "b"] = np.nan
df1.loc[1, "b"] = np.nan
```

```
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	NaN	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

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

	a	b	c	d
0	True	True	True	True
1	True	True	True	True
2	True	True	True	True

```
1.div(df1)
```

```
  File "C:\Users\smani\AppData\Loca
    1.div(df1)
     ^
SyntaxError: invalid syntax
```

Sorting

- To sort lexicographically by row or column label, use the sort_index method, which returns a new, sorted object
- you can sort by rows or columns by specifying axis key word

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

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

```
frame.sort_index(axis='index')
```

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

```
frame.sort_index(axis='columns')
```

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

Sorting

- The data is sorted in ascending order by default but can be sorted in descending order by the ascending key word
- sort_values is used to sort a Series and DataFrame
- Any missing values are sorted to the end of the Series by default
- Missing values can be sorted to the start instead by using the na_position argument to be first

```
frame.sort_index(axis="columns", ascending=False)
```

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

```
obj = pd.Series([4, 7, -3, 2])
obj
```

```
0    4
1    7
2   -3
3    2
dtype: int64
```

```
obj.sort_values()
```

```
2   -3
3    2
0    4
1    7
dtype: int64
```

```
obj.sort_values(na_position="first")
```

```
1    NaN
3    NaN
4   -3.0
5    2.0
```

Sorting

- While sorting a DataFrame, we can use the column name as the criteria to sort
- either pass a column name in the parentheses or multiple names in a column in side the parentheses

```
frame = pd.DataFrame(
    {"a": [-4, 7, 3, 2],
     "b": [0, 1, 0, 1]})
frame
```

	a	b
0	-4	0
1	7	1
2	3	0
3	2	1

```
frame.sort_values("b")
```

	a	b
0	-4	0
2	3	0
1	7	1
3	2	1

```
frame.sort_values(["a", "b"])
```

	a	b
0	-4	0
3	2	1
2	3	0
1	7	1

ranking

- Compute numerical data ranks 1 through n(the length)
- The smallest gets rank one and largest gets rank n
- Equal values are assigned a rank that is the average of the ranks of those values

```
obj = pd.Series([2,3,1])
print(obj.rank())
```

```
0    2.0
1    3.0
2    1.0
dtype: float64
```

```
obj = pd.Series([3,2,3,1])
print(obj.rank())
```

```
0    3.5
1    2.0
2    3.5
3    1.0
dtype: float64
```

ranking

- If same values appear more than once, Ranks can also be assigned according to the order in which they're observed in the data
- we can also rank in descending order
- ranks for DataFrame can be computed either for rows or columns

```
obj=pd.Series([3,2,3,1])
obj.rank(method='first')
```

```
0    3.0
1    2.0
2    4.0
3    1.0
dtype: float64
```

```
obj.rank(ascending=False)
```

```
0    1.5
1    3.0
2    1.5
3    4.0
dtype: float64
```

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

	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')
```

	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

Summary statistics

- numpy methods like sum(), mean(), cumsum(), can be applied by specifying axis labels

```
df=pd.DataFrame(
    np.random.randn(9).reshape((3,3)))
df
```

	0	1	2
0	0.893613	0.684894	-0.719420
1	0.249749	1.141725	1.926191
2	1.910010	0.149021	-1.757654

```
df.sum()
```

```
0    3.053371
1    1.975640
2   -0.550882
dtype: float64
```

```
df.sum(axis=0)
```

```
0     3.053371
1     1.975640
2    -0.550882
dtype: float64
```

```
df.sum(axis=1)
```

```
0     0.859087
1     3.317664
2     0.301377
dtype: float64
```

```
df.mean(axis=1)
```

```
0     0.286362
1     1.105888
2     0.100459
dtype: float64
```

```
df.cumsum()
```

	0	1	
0	0.893613	0.684894	-0.71942
1	1.143361	1.826619	1.20677
2	3.053371	1.975640	-0.55088

```
df.cumsum(axis=1)
```

	0	1	
0	0.893613	1.578506	0.85908
1	0.249749	1.391473	3.31766
2	1.910010	2.059031	0.30137

- idxmin and idxmax, return the index value where the minimum or maximum values are attained

```python
import pandas as pd
import numpy as np
obj=pd.DataFrame(np.random.
              randn(30).reshape((5,6)))
obj
```

	0	1	2	3	4	5
0	-0.528642	0.438604	-1.850550	-0.465469	-1.754782	-0.878637
1	-0.992590	-0.253095	-1.229630	1.249513	-0.929660	-0.469812
2	0.383950	-0.677413	0.096648	-0.521331	-1.078097	0.383043
3	2.341758	-1.092688	0.132866	0.352772	0.946163	0.990012
4	-0.589614	-0.436097	1.995528	3.191171	1.866934	-0.216534

```python
obj.idxmax()
```

```
0    3
1    0
2    4
3    4
4    4
5    3
dtype: int64
```

```python
obj.idxmax(axis=1)
```

```
0    1
1    3
2    0
3    0
4    3
dtype: int64
```

```python
obj.idxmax(axis=0)
```

Describe

```
obj.describe()
```

	0	1	2	3	4	5
count	5.000000	5.000000	5.000000	5.000000	5.000000	5.000000
mean	0.122972	-0.404138	-0.171028	0.761331	-0.189888	-0.038386
std	1.338654	0.566310	1.483189	1.537825	1.525303	0.734693
min	-0.992590	-1.092688	-1.850550	-0.521331	-1.754782	-0.878637
25%	-0.589614	-0.677413	-1.229630	-0.465469	-1.078097	-0.469812
50%	-0.528642	-0.436097	0.096648	0.352772	-0.929660	-0.216534
75%	0.383950	-0.253095	0.132866	1.249513	0.946163	0.383043
max	2.341758	0.438604	1.995528	3.191171	1.866934	0.990012

Exercise

Consider a column containing non-numeric data and then apply describe method to that. a) first apply to a non-numeric series b)a data frame with one column non-numeric

Summary statisitcs and other methods

`count`	Number of non-NA values
`describe`	Compute set of summary statistics
`min, max`	Compute minimum and maximum values
`argmin, argmax`	Compute index locations (integers) at which minimum or maximum value is obtained, respectively; not available on DataFrame objects
`idxmin, idxmax`	Compute index labels at which minimum or maximum value is obtained, respectively
`quantile`	Compute sample quantile ranging from 0 to 1 (default: 0.5)
`sum`	Sum of values
`mean`	Mean of values
`median`	Arithmetic median (50% quantile) of values
`mad`	Mean absolute deviation from mean value
`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

single observed value of dependent variable

mean of all values of independent variable

single observed value of independent variable

total count of sample values

$$Cov(X,Y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{n-1}$$

mean of all values of independent variable

population count minus one (Bessel's Correction)

$$Cor(X,Y) = \frac{Cov(X,Y)}{s_x s_y}$$

covariance

$$= \quad Cor(X,Y) = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2 (y_i - \bar{y})^2}}$$

standard deviations

Syntax

- series_name1.cov(series_name2)
- series_name1.corr(series_name2)
- df.cov() for covariance and df.corr() for corelation among columns of the data frame.

unique values, counts

- function is unique, which gives you an array of the unique values in a Series
- value_counts computes a Series containing value frequencies
- isin performs a vectorized set membership check

Covariance

```
obj[0].cov(obj[0])
```

0.8261131446271064

```
obj.cov()
```

	0	1	2	3	4	5
0	0.826113	0.101669	-0.610734	1.128580	-0.377262	-0.979280
1	0.101669	0.770697	0.475538	1.436232	-0.082309	0.466032
2	-0.610734	0.475538	1.565414	0.505303	0.374347	1.454724
3	1.128580	1.436232	0.505303	5.724417	0.946254	1.015754
4	-0.377262	-0.082309	0.374347	0.946254	1.410920	1.457691
5	-0.979280	0.466032	1.454724	1.015754	1.457691	2.544626

Correlation

```
obj[0].corr(obj[0])
```

```
1.0
```

```
obj.corr()
```

	0	1	2	3	4	5
0	1.000000	0.127417	-0.537054	0.518976	-0.349440	-0.675422
1	0.127417	1.000000	0.432941	0.683781	-0.078932	0.332784
2	-0.537054	0.432941	1.000000	0.168800	0.251889	0.728877
3	0.518976	0.683781	0.168800	1.000000	0.332959	0.266141
4	-0.349440	-0.078932	0.251889	0.332959	1.000000	0.769312
5	-0.675422	0.332784	0.728877	0.266141	0.769312	1.000000

isin function

```
obj = pd.Series(["c", "a", "d", "a", "a", "b", "b", "c", "c"])
uniques = obj.unique()
uniques
```

```
array(['c', 'a', 'd', 'b'], dtype=object)
```

```
obj.value_counts()
```

```
c    3
a    3
b    2
d    1
dtype: int64
```

```
mask=obj.isin(["b", "c"])
mask
```

```
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool
```

```
obj[mask]
```

```
0    c
5    b
6    b
7    c
8    c
```