Introduction to Pandas

- Basic Data Types
- Indexing and Selection
- Filtering

Import both `pandas` and `numpy` libraries.

In [1]:
```
import numpy as np
import pandas as pd

print(np.__version__)
print(pd.__version__)
```

```
1.18.5
1.0.5
```

1. Basic Data Types

Both NumPy and Pandas provide useful data types to work with data.

Numpy ndarray

The `np.random.rand(n)` function generates `n` number of random values uniformly distributed over `[0,1)` .

- The `np.random.seed(0)` fix the seed value to `0` so that the randomly generated values are reproducible.
- Generating of random numbers are much more efficient using Numpy.

Exercise

Generate 3 random numbers using Numpy.

In [2]:
```
np.random.seed(0)
r = np.random.rand(5)
r
```

Out[2]:
```
array([0.5488135 , 0.71518937, 0.60276338, 0.54488318, 0.4236548 ])
```

Pandas Series

Create a Panda Series using Numpy ndarray.

In [3]:
```
1 s = pd.Series(r)
2 print(type(s))
3 print(s)
```

```
<class 'pandas.core.series.Series'>
0    0.548814
1    0.715189
2    0.602763
3    0.544883
4    0.423655
dtype: float64
```

A Pandas Series can behave like a list too!

In [4]:
```
1 s[:3]
```

Out[4]:
```
0    0.548814
1    0.715189
2    0.602763
dtype: float64
```

Question:

Since both Numpy ndarray and Pandas Series behave like a list. Why do we need to use Panda Series?

- Pandas Series contains not only data but also an index, aka a label, for each record.

Set Index Names

We can change the index value.

In [5]:
```
1 s.index = ['a', 'b', 'c', 'd', 'e']
2 s
```

Out[5]:
```
a    0.548814
b    0.715189
c    0.602763
d    0.544883
e    0.423655
dtype: float64
```

Index can be set at creation.

In [6]:
```
1 s = pd.Series(r, index = ['a', 'b', 'c', 'd', 'e'])
```

Now we can access data using numeric index or text index.

In [7]:
```
1 s[1] == s['b']
```

Out[7]: True

Set Column Name

Besides index value, we can also assign a name to the column.

In [8]:
```
1 s.name = 'MyRandom'
2 s
```

Out[8]:
```
a    0.548814
b    0.715189
c    0.602763
d    0.544883
e    0.423655
Name: MyRandom, dtype: float64
```

Pandas DataFrame

Pandas Series can only contains 1 column of data. Pandas DataFrame allows multiple columns, which is like combining of multiple Pandas Series objects which have same index value.

Create a ndarray of 3 rows and 2 columns with random number.

In [9]:
```
1 r = np.random.rand(5,3)
2 r
```

Out[9]:
```
array([[0.64589411, 0.43758721, 0.891773  ],
       [0.96366276, 0.38344152, 0.79172504],
       [0.52889492, 0.56804456, 0.92559664],
       [0.07103606, 0.0871293 , 0.0202184 ],
       [0.83261985, 0.77815675, 0.87001215]])
```

Create a Pandas DataFrame from above ndarray.

In [10]:
```
df = pd.DataFrame(r)
df
```

Out[10]:

	0	1	2
0	0.645894	0.437587	0.891773
1	0.963663	0.383442	0.791725
2	0.528895	0.568045	0.925597
3	0.071036	0.087129	0.020218
4	0.832620	0.778157	0.870012

Set Index Labels

Update index labels of the dataframe.

In [11]:
```
df.index = ['a', 'b', 'c', 'd', 'e']
df
```

Out[11]:

	0	1	2
a	0.645894	0.437587	0.891773
b	0.963663	0.383442	0.791725
c	0.528895	0.568045	0.925597
d	0.071036	0.087129	0.020218
e	0.832620	0.778157	0.870012

Set Column Names

Update columns names of the dataframe.

In [12]:
```
df.columns = ['A', 'B', 'C']
df
```

Out[12]:

	A	B	C
a	0.645894	0.437587	0.891773
b	0.963663	0.383442	0.791725
c	0.528895	0.568045	0.925597
d	0.071036	0.087129	0.020218
e	0.832620	0.778157	0.870012

Rename Index and/or Name

You may use `dataframe.rename()` function to rename an index or a column.

- By default, `rename()` function returns a new DataFrame, i.e. it doesn't affect original DataFrame.
- To modify the original DataFrame directly, add parameter `inplace=True` .

In [13]:
```
df.rename(columns={'A':'A1'}, index={'a':'a1'})
```

Out[13]:

	A1	B	C
a1	0.645894	0.437587	0.891773
b	0.963663	0.383442	0.791725
c	0.528895	0.568045	0.925597
d	0.071036	0.087129	0.020218
e	0.832620	0.778157	0.870012

2. Indexing and Selection

Select Rows by Position

Pandas provides `iloc[R,C]` selects rows by positions.

The `iloc` accepts a list `row positions` , and optionally a list `column positions` .

```
df.iloc[row_positions, column_positions]
```

Select cell at 1st and 2nd row, including all columns.

In [14]:
```
df
```

Out[14]:

	A	B	C
a	0.645894	0.437587	0.891773
b	0.963663	0.383442	0.791725
c	0.528895	0.568045	0.925597
d	0.071036	0.087129	0.020218
e	0.832620	0.778157	0.870012

In [15]:
```
# df.iloc[row_position_list, col_position_list]
df.iloc[[0,1]]
```

Out[15]:

	A	B	C
a	0.645894	0.437587	0.891773
b	0.963663	0.383442	0.791725

Select first 2 rows and first 2 columns.

In [16]:
```
# df.iloc[row_position_list, col_position_list]
df.iloc[0:2, 0:2]
```

Out[16]:

	A	B
a	0.645894	0.437587
b	0.963663	0.383442

Select Rows by Label

Pandas provides `loc[ ]` function to select rows by labels.

The `loc` accepts a list `row indexers` which specifies row indexes, and a list `column_indexes` which specifies column names.

```
df.loc[row_indexers, column_indexers]
```

Get rows with label `b` and `c` .

In [17]:
```
# df.iloc[row_indexes_list]
df.loc[['b', 'c']]
```

Out[17]:

	A	B	C
b	0.963663	0.383442	0.791725
c	0.528895	0.568045	0.925597

Select a subset of the dataframe to include row `b` and `c` , column `B` and `C` .

In [18]:
```
# # df.loc[row_indexes_list, col_indexes_list]
df.loc[['b','c'], ['B','C']]
```

Out[18]:

	B	C
b	0.383442	0.791725
c	0.568045	0.925597

If row indexer and column indexer are a single value instead of a list, the result is a cell value instead of a DataFrame.

In [19]:
```
result = df.loc[['a'], ['B']]
result
```

Out[19]:

	B
a	0.437587

In [20]:
```
result = df.loc['a', 'B']
result
```

Out[20]: 0.4375872112626925

Select Columns

We can now select columns by respective column names.

In [21]:
```
# df.loc[row_indexes_list, col_indexes_list]

df.loc[:, ['A', 'B']]
```

Out[21]:

	A	B
a	0.645894	0.437587
b	0.963663	0.383442
c	0.528895	0.568045
d	0.071036	0.087129
e	0.832620	0.778157

Here is a shortcut to select multiple columns using `[]` .

In [22]:
```
result = df[['A', 'B']]
type(result)
```

Out[22]: pandas.core.frame.DataFrame

Each column is in fact a Pandas Series.

In [23]:
```
result = df['A']
type(result)
```

Out[23]:
```
pandas.core.series.Series
```

3. Filtering

Max and Min Value

The `max()` and `min()` functions return max and min values of each column.

In [24]:
```
print(df)
# df.max()
df.min()
```

```
          A         B         C
a  0.645894  0.437587  0.891773
b  0.963663  0.383442  0.791725
c  0.528895  0.568045  0.925597
d  0.071036  0.087129  0.020218
e  0.832620  0.778157  0.870012
```

Out[24]:
```
A    0.071036
B    0.087129
C    0.020218
dtype: float64
```

The `idxmax()` and `idxmin()` functions returns the row label whose row value is max or min value.

- To get the row poistion instead of row label, use `argmax()` and `agrmin()` instead.

In [25]:
```
# df.idxmin()
b = df['A'].idxmin()
b
```

Out[25]:
```
'd'
```

In [26]:
```
df.loc[b]
```

Out[26]:
```
A    0.071036
B    0.087129
C    0.020218
Name: d, dtype: float64
```

Exercise:

Find the row whose `B` column is the minimum value of the column.

In [27]:
```
i = df['B'].argmin()
df.iloc[i]
```

Out[27]:
```
A    0.071036
B    0.087129
C    0.020218
Name: d, dtype: float64
```

Filtering

Rows in dataframe can be filtered by list of boolean values.

In [28]:
```
f = [True, False, True, False, False]
df[f]
```

Out[28]:

	A	B	C
a	0.645894	0.437587	0.891773
c	0.528895	0.568045	0.925597

To check if rows in dataframe fulfills certain condiction, we can use comparison expression with the dataframe.

For example, which rows in column `A` value are less than 0.5?

In [29]:
```
m = df['A'] < 0.5
print(m)
```

```
a    False
b    False
c    False
d     True
e    False
Name: A, dtype: bool
```

We can continue to use above value to filter dataframe.

In [30]:
```
# df[m]
df[ df['A'] < 0.5 ]
```

Out[30]:

	A	B	C
d	0.071036	0.087129	0.020218

Exercise:

Show the rows whose column `B` value is greater than `0.7` ?

In [31]:
```
1 n = df['B'] > 0.7
2 df[n]
```

Out[31]:

	A	B	C
e	0.83262	0.778157	0.870012

Join Multiple Conditions

Multiple Conditions can be joined together using `&` (AND) and `|` (OR) operators.

Exercise:

Show the rows whose both column `A` and column `B` values are greater than `0.5` ?

In [32]:
```
1 df[ (df['A'] > 0.5) & (df['B'] > 0.5) ]
```

Out[32]:

	A	B	C
c	0.528895	0.568045	0.925597
e	0.832620	0.778157	0.870012