Pandas Introduction to Data Structures

1. What is Pandas?

Python library for data analysis

- Provides fast and flexible data structures designed to work with tabular data
- Built on top of NumPy
- Part of the SciPy ecosystem (Scientific Computing Tools for Python)
  - Integrated well with other Python packages
  - SciPy & StatsModel, Matplotlib & Plotly, Scikit-learn

Key Pandas Features

- Intuitive data format
- Easy data transformations
- Data visualization
- Ideal tools for typical data engineering task cycle
  - Munging, Cleaning, Analyzing and Modeling data
  - Organizing result for visualization or tabular display

Import Pandas package and check its version.

In [1]:

```
import pandas as pd
pd.__version__
```

Out[1]:

```
'0.25.3'
```

Data Structure

- Pandas supports up to two-dimentions DataFrame
- 1D objects are called Series.
- 2D objects are called DataFrame.
- The structure is Rows and Columns.

The basics

Pandas documentation https://pandas.pydata.org/pandas-docs/stable/getting_started/basics.html (https://pandas.pydata.org/pandas-docs/stable/getting_started/basics.html)

2. Pandas Series

Pandas Series is a one-dimensional array with axis labels.

Constructing Series Objects

Series object can be created using following constrcutor, where data can be a list, dictionary or another Series.

```
pandas.Series(data, index)
```

Create from List

Without specifying index, Series will be assigned with 0-based numeric index value.

In [2]:

```
s = pd.Series(list('abcde'))
s
```

Out[2]:

```
0    a
1    b
2    c
3    d
4    e
dtype: object
```

In [3]:

```
s.index
```

Out[3]:

```
RangeIndex(start=0, stop=5, step=1)
```

In [4]:

```
s.values
```

Out[4]:

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

Specify Index

A Series index can be specified. Following command is the same as

```
s1 = pd.Series(range(100,105))
s1.index = list('abcde')
s2 = pd.Series(range(100,105), index = list('abcde'))
```

In [5]:

```
s = pd.Series(range(100,105))
s.index = list('abcde')
s
```

Out[5]:

```
a    100
b    101
c    102
d    103
e    104
dtype: int64
```

Selecting Items

Items in a Series can be selected by position, which supports both single item indexing and slicing.

In [6]:

```
s[0]
```

Out[6]:

```
100
```

In [7]:

```
s[3:]
```

Out[7]:

```
d    103
e    104
dtype: int64
```

Items in a Series can also be selected by label.

In [8]:

```
s['a']
```

Out[8]:

```
100
```

In [9]:

```
s[['a', 'b']]
```

Out[9]:

```
a    100
b    101
dtype: int64
```

Specialized Dictionary

A Series object is like a Dictionary object, which maps keys (index) to values (data). But with following differences:

- Items in Series is ordered
- Series has a fixed-length
- Keys (index) in Series don't have to be unique

In fact, a Series object can be created from a dictionary.

In [10]:

```
_dict = {'a':'apple', 'b':'banana', 'c':'cherry', 'd':'donut'}
fruits = pd.Series(_dict)
fruits
```

Out[10]:

```
a     apple
b    banana
c    cherry
d     donut
dtype: object
```

In [11]:

```
fruits['d'] = 'apricots'
fruits
```

Out[11]:

```
a       apple
b      banana
c      cherry
d    apricots
dtype: object
```

In [12]:

```
fruits = fruits.rename({'d':'a'})
fruits
```

Out[12]:

```
a       apple
b      banana
c      cherry
a    apricots
dtype: object
```

In [13]:

```
fruits['a']
```

Out[13]:

```
a       apple
a    apricots
dtype: object
```

Filtering Data

Similiar to NumPy array, data in Series can be filtered by boolean values.

Let's generate 10 random integers between 100 and 110. Use it to create a Series object.

In [14]:

```
import numpy as np
np.random.seed(0)

# Generate 10 random integer between 100 and 110
nums = np.random.randint(100,110,10)
# Create Series using nums
s = pd.Series(nums)
s
```

Out[14]:

```
0    105
1    100
2    103
3    103
4    107
5    109
6    103
7    105
8    102
9    104
dtype: int32
```

We can create boolean array where corresponding value in Series is greater than 105.

In [15]:

```
# List of boolean with number > 105
b = nums > 105
b
```

Out[15]:

```
array([False, False, False, False,  True,  True, False, False, False,
       False])
```

Filter Series using boolean values. Following statement gives same output.

```
s[s>105]
```

In [16]:

```
s[b]
# s[s>105]
```

Out[16]:

```
4    107
5    109
dtype: int32
```

Filtering by Multiple Conditions

Multiple conditions can be combined using `&` (AND) and `|` (OR) operators.

- Find values in Series which can be divided by both 2 and 3.
- Find values in Series which can be divided by either 2 or 3.

In [17]:

```
s[(s%2==0) & (s%3==0)]
# s[(s%2==0) | (s%3==0)]
```

Out[17]:

```
8    102
dtype: int32
```

Missing Data and Auto Alignment

Pandas can accomodate incomplete data. Missing data will have a value of `NaN` , i.e. Not-a-Number.

Data will be automatically aligned by their index values.

In [18]:

```
s0 = pd.Series(range(100,105), index=list('bcdfg'))
s0
```

Out[18]:

```
b    100
c    101
d    102
f    103
g    104
dtype: int64
```

Create another Series with an existing Series object and specifying new index.

- Item, whose index does not exists in original Series, is set to `NaN`
- Item, whose index does not exists in new Series, is dropped.

For example:

- Items with index 'a' and 'e' are assigned with `NaN` .
- Items with index 'g' is dropped.

In [19]:

```
s = pd.Series(s0, index=list('abcdef'))
s
```

Out[19]:

```
a      NaN
b    100.0
c    101.0
d    102.0
e      NaN
f    103.0
dtype: float64
```

3. Pandas DataFrames

Pandas DataFrame is a 2-dimensional tabular data structure, which contains rows and columns.

- Columns can have different types.
- Columns can be added and removed.
- Rows and columns are indexed anc can be labeled.

Reference: https://www.geeksforgeeks.org/python-pandas-dataframe/

Create DataFrame

A pandas DataFrame can be created using various inputs. All columns must be equal-length.

```
pandas.DataFrame( data, index, columns)
```

It can be considered as dictionary of Series/Lists with shared row index.

- Data is commonly passed in dictionary form, whose keys will become column labels

Create from Lists as Columns

In [20]:

```
# Create a dictionary
d = {'col1':range(50,55), 'col2':range(60,65), 'col3':range(70,75)}

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

Out[20]:

	col1	col2	col3
0	50	60	70
1	51	61	71
2	52	62	72
3	53	63	73
4	54	64	74

By default, Both DataFrame's row and column labels are integer values starting from 0.

In [21]:

```
print(df.columns)
print(df.index)
```

```
Index(['col1', 'col2', 'col3'], dtype='object')
RangeIndex(start=0, stop=5, step=1)
```

Create from Series as Columns

DataFrame can also be create from existing Series objects.

In [22]:

```
# Create a dictionary
s1 = pd.Series(range(50,55))
s2 = pd.Series(range(60,65))
s3 = pd.Series(range(70,75))

d = {'col1':s1, 'col2':s2, 'col3':s3}

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

Out[22]:

	col1	col2	col3
0	50	60	70
1	51	61	71
2	52	62	72
3	53	63	73
4	54	64	74

Create from 2D Lists as Rows

DataFrame objct can also be created using rows of data.

Rows of data are passed as a nested list object.

In [23]:

```
pd.DataFrame([range(100,110), range(110,120)])
```

Out[23]:

	0	1	2	3	4	5	6	7	8	9
0	100	101	102	103	104	105	106	107	108	109
1	110	111	112	113	114	115	116	117	118	119

Select Column(s)

Columns can be retrieved as Series

- dictionary notation
- attribute notation

In [24]:

```
df['col2']
```

Out[24]:

```
0    60
1    61
2    62
3    63
4    64
Name: col2, dtype: int64
```

In [25]:

```
df.col2
```

Out[25]:

```
0    60
1    61
2    62
3    63
4    64
Name: col2, dtype: int64
```

To select multiple columns, use list of columns labels using dictionary notation.

In [26]:

```
df[['col1', 'col3']]
```

Out[26]:

	col1	col3
0	50	70
1	51	71
2	52	72
3	53	73
4	54	74

Add Column(s)

New columns can be easily added

- direct assignment
- computation from other columns

Note: Columns cannot be added using attribute notation!

In [27]:

```
# Add new column by direct assignment
df['col4'] = np.random.randint(50,100,5)
df
```

Out[27]:

	col1	col2	col3	col4
0	50	60	70	73
1	51	61	71	56
2	52	62	72	74
3	53	63	73	74
4	54	64	74	62

In [28]:

```
# Compute new column from existing columns
df['col5'] = df.col1 + df.col2
df
```

Out[28]:

	col1	col2	col3	col4	col5
0	50	60	70	73	110
1	51	61	71	56	112
2	52	62	72	74	114
3	53	63	73	74	116
4	54	64	74	62	118

Add Column of Same Value

NumPy's broadcasting feature make it easy to add a new column with same value.

In [29]:

```
df['col6'] = 99
df
```

Out[29]:

	col1	col2	col3	col4	col5	col6
0	50	60	70	73	110	99
1	51	61	71	56	112	99
2	52	62	72	74	114	99
3	53	63	73	74	116	99
4	54	64	74	62	118	99

Auto Alignment

Column can be added by a Series, where indexes will be automatically aligned.

In [30]:

```
df['col5'] = pd.Series([80,90,80,90], index=[0,2,3,5])
print(df)
```

```
   col1  col2  col3  col4  col5  col6
0    50    60    70    73  80.0    99
1    51    61    71    56   NaN    99
2    52    62    72    74  90.0    99
3    53    63    73    74  80.0    99
4    54    64    74    62   NaN    99
```

Reindexing

Reindexing will create a new object with data conformed to the new index.

- Rows not in new index will be dropped.
- Rows not in existing index will have values of `NaN` .

In [31]:

```
df2 = df.reindex(range(1, 10))
df2
```

Out[31]:

	col1	col2	col3	col4	col5	col6
1	51.0	61.0	71.0	56.0	NaN	99.0
2	52.0	62.0	72.0	74.0	90.0	99.0
3	53.0	63.0	73.0	74.0	80.0	99.0
4	54.0	64.0	74.0	62.0	NaN	99.0
5	NaN	NaN	NaN	NaN	NaN	NaN
6	NaN	NaN	NaN	NaN	NaN	NaN
7	NaN	NaN	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN	NaN	NaN
9	NaN	NaN	NaN	NaN	NaN	NaN

Delete a Column

To delete a column, you can use `pop()` or `drop()` functions. But they are different.

- `pop()` function modify the DataFrame object directly.
- `drop()` function returns a new object, and you must specify `axis=1` which is referred to column.

In [32]:

```
# df2.pop('col1')
df2 = df2.drop('col4', axis=1)
df2
```

Out[32]:

	col1	col2	col3	col5	col6
1	51.0	61.0	71.0	NaN	99.0
2	52.0	62.0	72.0	90.0	99.0
3	53.0	63.0	73.0	80.0	99.0
4	54.0	64.0	74.0	NaN	99.0
5	NaN	NaN	NaN	NaN	NaN
6	NaN	NaN	NaN	NaN	NaN
7	NaN	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN	NaN
9	NaN	NaN	NaN	NaN	NaN

Change Index Column

In [33]:

```
df2 = df.copy()
df2['col0'] = list('abcde')
df2
```

Out[33]:

	col1	col2	col3	col4	col5	col6	col0
0	50	60	70	73	80.0	99	a
1	51	61	71	56	NaN	99	b
2	52	62	72	74	90.0	99	c
3	53	63	73	74	80.0	99	d
4	54	64	74	62	NaN	99	e

In [34]:

```
df2 = df2.set_index('col0')
df2
```

Out[34]:

	col1	col2	col3	col4	col5	col6
col0						
a	50	60	70	73	80.0	99
b	51	61	71	56	NaN	99
c	52	62	72	74	90.0	99
d	53	63	73	74	80.0	99
e	54	64	74	62	NaN	99

Row Selection & Slicing

Rows can be selected using either `iloc[]` or `loc[]` .

- `iloc[]` function accepts row positions
- `loc[]` function accepts labels

In [35]:

```
df2.iloc[1]
```

Out[35]:

```
col1    51.0
col2    61.0
col3    71.0
col4    56.0
col5     NaN
col6    99.0
Name: b, dtype: float64
```

In [36]:

```
df2.iloc[:2]
```

Out[36]:

	col1	col2	col3	col4	col5	col6
col0						
a	50	60	70	73	80.0	99
b	51	61	71	56	NaN	99

In [37]:

```
df2.loc['a']
```

Out[37]:

```
col1    50.0
col2    60.0
col3    70.0
col4    73.0
col5    80.0
col6    99.0
Name: a, dtype: float64
```

In [38]:

```
df2.loc[['a', 'b']]
```

Out[38]:

	col1	col2	col3	col4	col5	col6
col0						
a	50	60	70	73	80.0	99
b	51	61	71	56	NaN	99

Add Rows

Add new rows to a dataFrame can be done by `append()` function.

In [39]:

```
df3 = pd.DataFrame([[88,88,88,88]], columns=['col1','col2', 'col3', 'col4'], index=['f
df3
```

Out[39]:

	col1	col2	col3	col4
f	88	88	88	88

In [40]:

```
df4 = df2.append(df3)
df4
```

```
C:\Users\zqi2\AppData\Local\Continuum\anaconda3\lib\site-packages\pandas
\core\frame.py:7138: FutureWarning: Sorting because non-concatenation axi
s is not aligned. A future version
of pandas will change to not sort by default.

To accept the future behavior, pass 'sort=False'.

To retain the current behavior and silence the warning, pass 'sort=True'.

  sort=sort,
```

Out[40]:

	col1	col2	col3	col4	col5	col6
a	50	60	70	73	80.0	99.0
b	51	61	71	56	NaN	99.0
c	52	62	72	74	90.0	99.0

Delete Row(s)

Rows can be deleted by `drop()` function using its label.

- By default, `drop()` function has parameter `axis=0` which refers to row.
- `drop()` function creates a new object.

In [41]:

```
df5 = df4.drop('a')
df5
```

Out[41]:

	col1	col2	col3	col4	col5	col6
b	51	61	71	56	NaN	99.0
c	52	62	72	74	90.0	99.0
d	53	63	73	74	80.0	99.0
e	54	64	74	62	NaN	99.0
f	88	88	88	88	NaN	NaN