Pandas (*)

Pandas (Panel Data) is an open source Python library for data analysis (*).

To describe Pandas we use the Jupyter Notebook editor, part of Anaconda installation

To start using Pandas we import the following modules:

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

In [2]: import pandas as pd
```

It is usual to refer to **numpy** module as **np** and **pandas** module as **pd**

Pandas has the following data structures:

Series - is a one-dimensional array that can hold any data type (int, float, string, objects, etc.). It has an axis label, known as index.

DataFrame - is a two-dimensional data structure where each column can have different data types. The DataFrame concept is similar to a spreadsheet or SQL table.

(*) https://pandas.pydata.org/pandas-docs/stable/user_guide/index.html

Pandas – data structures - Series

To create a Series in Pandas: s = pd.Series(**data**, index=**index**)

data can be:

- a Python dict
- a Python list
- a ndarray
- a scalar value

index is a list of axis labels:

- if data is an array, **index** must be of same length
- if there is no **index**, one will be created [0,…,len(data)-1]
- **index** can have non-unique values
- when data is a dict and there is no index the index will be the dict keys

Pandas - Create a Series - examples

Import NumPy and Pandas modules

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

Creates a Series from a Python list.
If no index is passed one will be created by default

```
In [2]: s1 = pd.Series(['antonio','joana','carlos','barbara'])
        s1
```

```
Out[2]: 0    antonio
        1      joana
        2     carlos
        3    barbara
        dtype: object
```

```
In [3]: s1.index
```

```
Out[3]: RangeIndex(start=0, stop=4, step=1)
```

Creates a Series from an ndarray using NumPy randn function
(samples from a standard normal distribution)

```
In [4]: s2 = pd.Series(np.random.randn(4), index = ["a","b","c","d"])
        s2
```

```
Out[4]: a    1.558118
        b    0.276449
        c    0.406288
        d   -1.285551
        dtype: float64
```

```
In [5]: s2.index
```

```
Out[5]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

Creates a Series from a dict.
If no index is passed dict keys will be the index

```
In [6]: s3 = pd.Series({'b':20, 'a':30, 'c':15})
        s3
```

```
Out[6]: b    20
        a    30
        c    15
        dtype: int64
```

Pandas alligns the values by the indexes.
It uses NumPy np.nan (Not a Number) if there is no value.

```
In [7]: d = {'a':10,'b':20,'c':30}
        s4 = pd.Series(d, index=['b','c','d','a'])
        s4
```

```
Out[7]: b    20.0
        c    30.0
        d     NaN
        a    10.0
        dtype: float64
```

Creates a Series from a scalar value

```
In [8]: s5 = pd.Series(5, index = ['a','b','c'])
        s5
```

```
Out[8]: a    5
        b    5
        c    5
        dtype: int64
```

Pandas Series - examples

Series acts very similarly to a ndarray and can be used as a valid argument to most NumPy functions

A Series element can be accessed as in a ndarray

```
In [9]: s2[0]
Out[9]: 1.5581183754618142
```

Slicing can be used but includes the index

```
In [10]: s2[:3]
Out[10]: a    1.558118
         b    0.276449
         c    0.406288
         dtype: float64
```

A list of boolean can be used to access the values

```
In [11]: s2[s2 > s2.mean()]
Out[11]: a    1.558118
         b    0.276449
         c    0.406288
         dtype: float64
```

A list of numeric indexes can be used to access the values

```
In [12]: s2[[3,0,1]]
Out[12]: d   -1.285551
         a    1.558118
         b    0.276449
         dtype: float64
```

Series can be used with most NumPy functions

```
In [13]: np.exp(s2)
Out[13]: a    4.749874
         b    1.318440
         c    1.501235
         d    0.276498
         dtype: float64
```

Series has a dtype like a ndarray

```
In [14]: s2.dtype
Out[14]: dtype('float64')
```

A Pandas array can be obtained without index

```
In [15]: s2.array
Out[15]: <PandasArray>
         [1.558118, 0.276449, 0.406288, -1.285551]
         Length: 4, dtype: float64
```

A Series can be converted to a NumPy array

```
In [16]: s2.to_numpy()
Out[16]: array([ 1.558118,  0.276449,  0.406288, -1.285551])
```

Pandas Series - examples

Series is like a fixed-size dict in that you can access the values by index label

```
In [17]: s2["a"]
Out[17]: 1.558118

In [18]: s2["b"] = 20

In [19]: s2
Out[19]: a     1.558118
         b    20.000000
         c     0.406288
         d    -1.285551
         dtype: float64

In [20]: "c" in s2
Out[20]: True
```

Operations between Series automatically align the data based on labels

```
In [21]: s2[1:] + s2[:-1]
Out[21]: a          NaN
         b    40.000000
         c     0.812576
         d          NaN
         dtype: float64
```

The result of an operation between unaligned Series will have the union of the indexes involved

Series can have a name attribute

```
In [22]: s6 = pd.Series(np.random.randn(4), name = 'something')
         s6
Out[22]: 0   -1.325993
         1   -0.937542
         2   -0.524005
         3    0.710252
         Name: something, dtype: float64

In [23]: s6.name
Out[23]: 'something'
```

that can be renamed

```
In [24]: s6.rename('different',inplace=True)
         s6
Out[24]: 0   -1.325993
         1   -0.937542
         2   -0.524005
         3    0.710252
         Name: different, dtype: float64

In [25]: s6.name
Out[25]: 'different'
```

Pandas – data structures - DataFrame

A DataFrame accepts different kind of inputs:

- Dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray
- Structured or record ndarray
- A Series
- Another DataFrame

The input can optionally have index (row labels) and columns (column labels) arguments.

If axis labels are not used, they will be built based on common sense rules

Pandas – Create a DataFrame - examples

A DataFrame can be created from a dict of Series or dicts

In [26]:
```
d = {
    "one": pd.Series([1.0, 2.0, 3.0], index=["a", "b", "c"]),
    "two": pd.Series([1.0, 2.0, 3.0, 4.0], index=["a", "b", "c", "d"]),
}
df = pd.DataFrame(d)
df
```

Out[26]:

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0

Values are alligned by the index and NaN is assigned for no existing values

In [27]:
```
pd.DataFrame(d, index=["d", "b", "a"])
```

Out[27]:

	one	two
d	NaN	4.0
b	2.0	2.0
a	1.0	1.0

If columns is passed along with a dict of data
the passed columns override the keys in the dict.

In [28]:
```
pd.DataFrame(d, index=["d", "b", "a"], columns=["two", "three"])
```

Out[28]:

	two	three
d	4.0	NaN
b	2.0	NaN
a	1.0	NaN

The row and column labels can be accessed respectively
by accessing the index and columns attributes

In [29]:
```
df.index
```

Out[29]:
```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

In [30]:
```
df.columns
```

Out[30]:
```
Index(['one', 'two'], dtype='object')
```

Create a DataFrame from dict of ndarrays / lists

In [31]:
```
d = {"one": [1.0, 2.0, 3.0, 4.0], "two": [4.0, 3.0, 2.0, 1.0]}
```

If no index is passed the index will be range(n), n is the array length

In [32]:
```
pd.DataFrame(d)
```

Out[32]:

	one	two
0	1.0	4.0
1	2.0	3.0
2	3.0	2.0
3	4.0	1.0

The arrays and index must all be the same length

In [33]:
```
pd.DataFrame(d, index=["a", "b", "c", "d"])
```

Out[33]:

	one	two
a	1.0	4.0
b	2.0	3.0
c	3.0	2.0
d	4.0	1.0