Introduction to Data Structures

Pandas
in Python

Begin from the Beginning

- Pandas is used typically along with numpy

```
import numpy as np
import pandas as pd
```

- Two important data structures

  `pd.Series, pd.DataFrame`

- The data are labeled, and the link between will not be broken unless explicitly done so. That is the *data alignment is intrinsic*.

Series

- Series is a one-dimensional labeled array.
- Holds any data type (integers, strings, floating point numbers, Python objects, etc.).
- The axis labels are collectively referred to as the index.
- The basic method to create a Series is to call:

```
s = pd.Series(data, index=index)
```

- data can be python dict, ndarray, scalar(like 5)
- index – list of axis labels

Series from ndarray

- If data is an ndarray, index must be the same length as data.
- If no index is passed, one will be created having values [0, ..., len(data) – 1].

```
s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
```

```
print(s)
Out[4]:
a    0.469112
b   -0.282863
c   -1.509059
d   -1.135632
e    1.212112
dtype: float64
```

```
pd.Series(np.random.randn(5))
Out[6]:
0   -0.173215
1    0.119209
2   -1.044236
3   -0.861849
4   -2.104569
dtype: float64
```

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

Series from dict

```
In [7]: d = {'b': 1, 'a': 0, 'c': 2}
```

#Series index will be ordered by the dict’s insertion order, in Python version >= 3.6 and Pandas version >= 0.23.

```
In [8]: pd.Series(d)
Out[8]:
b    1
a    0
c    2
dtype: int64
.
```

```
In [11]: pd.Series(d, index=['b', 'c',
'd', 'a'])
Out[11]:
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64


.
```

Series from scalar

- If data is a scalar value, an index must be provided. The value will be repeated to match the length of index.

```
In [12]: pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
Out[12]:
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
```

What is Series?

- Series acts very similarly to a ndarray, and is a valid argument to most NumPy functions.
- Operations such as slicing will also slice the index.

```
s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
```

```
In [13]: s[0]
Out[13]: 0.4691122999071863
In [14]: s[:3]
Out[14]:
a    0.469112
b   -0.282863
c   -1.509059
dtype: float64
In [15]: s[s > s.median()]
Out[15]:
a    0.469112
e    1.212112
dtype: float64
```

```
s[[4, 3, 1]]
Out[16]:
e    1.212112
d   -1.135632
b   -0.282863
dtype: float64

In [17]:
np.exp(s)
```

```
np.exp(s)
Out[17]:
a    1.598575
b    0.753623
c    0.221118
d    0.321219
e    3.360575
dtype: float64
```

Backing store in a Series

```
s.array # array backing a Series

Out:

<PandasArray>

[ 0.4691122999071863, -0.2828633443286633, -
1.5090585031735124,

 -1.1356323710171934,  1.2121120250208506]

Length: 5, dtype: float64

In [20]: s.to_numpy() #ndarray backing a series

Out[20]: array([ 0.4691, -0.2829, -1.5091, -
1.1356,  1.2121])
```

Series is dict like

```
In [21]: s['a']
Out[21]: 0.4691122999071863
```

```
In [22]: s['e'] = 12.

In [23]: s
Out[23]:
a     0.469112
b    -0.282863
c    -1.509059
d    -1.135632
e    12.000000
dtype: float64

In [24]: 'e' in s
Out[24]: True

In [25]: 'f' in s
Out[25]: False
```

Using the get method, a missing label will return None or specified default:

```
s['f']
KeyError: 'f'

In [26]: s.get('f')

In [27]: s.get('f', np.nan)
Out[27]: nan
```

Vectorized operations and label alignment with Series

s + s	s * 2	np.exp(s)	s[1:] + s[:-1]
Out[28]:	Out[29]:	Out[30]:	Out[31]:
a 0.938225	a 0.938225	a 1.598575	a NaN
b -0.565727	b -0.565727	b 0.753623	b -0.565727
c -3.018117	c -3.018117	c 0.221118	c -3.018117
d -2.271265	d -2.271265	d 0.321219	d -2.271265
e 24.000000	e 24.000000	e 162754.791419	e NaN
dtype: float64	dtype: float64	dtype: float64	dtype: float64

A key difference between Series and ndarray is that operations between Series automatically align the data based on label.

Thus, you can write computations without giving consideration to whether the Series involved have the same labels.

Naming a Series

- Series can also have a name attribute:

```
s = pd.Series(np.random.randn(5), name='something')
s
Out[33]:
0   -0.494929
1    1.071804
2    0.721555
3   -0.706771
4   -1.039575
Name: something, dtype: float64

In [34]: s.name
Out[34]: 'something'
```

```
In [35]: s2 = s.rename("someotherthing")
In [36]: s2.name
Out[36]: 'someotherthing'

Note that s and s2 refer to different objects.
```

Aggregation

.sum: Returns the result of adding all values in a Series together.

.product: Returns the result of multiplying all values in a Series.

.mean: Calculates the average value by adding all values and dividing by the total rows.

.median: Returns the midpoint in a numerical data set.

.max: Finds the largest number in a Series.

.min: Finds the smallest number in a Series.

```
num_series.agg(['sum','product','mean','median','max','min'])
```

A few more...

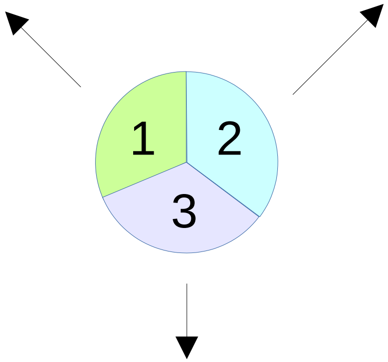
- num_series.fillna(0,inplace=True)
- num_series.dropna(inplace=True)
- num_series.iloc[2]
- num_series.iloc[0:3]
- num_series.loc["pos2"]
- num_series.loc[["pos4", "pos2"]]
- num_series.sort_index(inplace = True)
- name_series.sort_values(ascending = False, inplace = True)

DataFrame from dict of Series

```
d = {'one': pd.Series([1, 2, 3],    index=['a', 'b', 'c']),
     'two': pd.Series([1, 2, 3, 4], index=['a', 'b', 'c',
'd'])}
```

```
df = pd.DataFrame(d)    pd.DataFrame(d, index=['d', 'b', 'a'])
```

```
In [39]: df
Out[39]:
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0
```



```
Out[40]:
one  two
d  NaN  4.0
b  2.0  2.0
a  1.0  1.0
```

```
pd.DataFrame(d, index=['d', 'b', 'a'],
                columns=['two', 'three'])

Out[41]:
   two three
d  4.0   NaN
b  2.0   NaN
a  1.0   NaN
```

DataFrame from dict of ndarrays

- The ndarrays must all be the same length.
- If an index is passed, it must clearly also be the same length as the arrays.
- If no index is passed, the result will be range(n), where n is the array length.

```
d = {'one': [1., 2., 3., 4.],
     'two': [4., 3., 2., 1.]}
pd.DataFrame(d)         pd.DataFrame(d, index=['a', 'b', 'c', 'd'])
Out[45]:                Out[46]:
   one  two                one  two
0  1.0  4.0             a  1.0  4.0
1  2.0  3.0             b  2.0  3.0
2  3.0  2.0             c  3.0  2.0
3  4.0  1.0             d  4.0  1.0
```

DataFrame from record array

```
data = np.zeros((2, ), dtype=[('A', 'i4'), ('B', 'f4'), ('C', 'a10')])
data[:] = [(1, 2., 'Hello'), (2, 3., "World")]
```

```
pd.DataFrame(data)
Out[49]:
   A    B          C
0  1  2.0  b'Hello'
1  2  3.0  b'World'
```

```
pd.DataFrame(data, index=['first', 'second'])
Out[50]:
        A    B          C
first   1  2.0  b'Hello'
second  2  3.0  b'World'
```

```
pd.DataFrame(data, columns=['C', 'A', 'B'])
Out[51]:
          C  A    B
0  b'Hello'  1  2.0
1  b'World'  2  3.0
```

DataFrame is not intended to work exactly like a 2-dimensional NumPy ndarray.

DataFrame from list of dicts

```
In [52]: data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

pd.DataFrame(data2)   pd.DataFrame(data2, index=['first', 'second'])
Out[53]:              Out[54]:
   a   b     c                a   b     c
0  1   2   NaN        first   1   2   NaN
1  5  10  20.0        second  5  10  20.0

pd.DataFrame(data2, columns=['a', 'b'])
Out[55]:
   a   b
0  1   2
1  5  10
```

DataFrame from dict of objects

```
In [9]: df2 = pd.DataFrame({'A': 1.,
   ...:                     'B': pd.Timestamp('20130102'),
   ...:                     'C': pd.Series(1, index=list(range(4)), dtype='float32'),
   ...:                     'D': np.array([3] * 4, dtype='int32'),
   ...:                     'E': pd.Categorical(["test", "train", "test", "train"]),
   ...:                     'F': 'foo'})
   ...:
```

```
In [10]: df2
Out[10]:
     A          B    C  D      E    F
0  1.0 2013-01-02  1.0  3   test  foo
1  1.0 2013-01-02  1.0  3  train  foo
2  1.0 2013-01-02  1.0  3   test  foo
3  1.0 2013-01-02  1.0  3  train  foo
```

```
In [11]: df2.dtypes
Out[11]:
A           float64
B    datetime64[ns]
C           float32
D             int32
E          category
F            object
dtype: object
```

DataFrame from Series

- The result will be a DataFrame with the same index as the input Series, and with one column whose name is the original name of the Series (only if no other column name provided).