L-3

Pandas

pandas

Import numpy as np
Import pandas as pd

data = pd.Series([0.25, 0.5, 0.75, 1.0])

data

put
0 0.25
1 0.50
2 0.75
3 1.00

-> data.values
array([0.25, 0.5, 0.75, 1.])

-> data.index
RangeIndex(start=0, stop=4, step=1)

-> data = pd.Series([1,2,3,4], index=['a','b','c','d'])
-> data.values
-> data.index > Index values are a, b, c, d.

-> Pandas are capable for non contiguous or nonsequential indices.

→ Population_dict = {'California': 32222,
'Texas': 1234,
'Newyork': 34222,
'Kerala': 349999}.

Population = Pd.Series(Population_dict).
population

→ Data can be a scalar, which is repeated to fill the specified index

→ pd.series(5) index = [100, 200, 300].

output
100 5
200 5
300 5.

Data can be a dictionary, in which index denotes the sorted dictkeys

→ Pd.Series({2:'a', 1:'b', 3:'c'})

output
1 b
2 a
3 c.

Imp in pandas

Series → one-D Array → Horizontal indices.

Dataframe → 2-D Array → both Horizontal row indices, Horizontal column names

-> area_dict = {'California': 32432, 'Texas': 695662,
'Newyork': 294532, 'Florida': 39454,
'Illinois': 149995}

-> area = pd.Series(area_dict)

Creating a 2-D object containing this info using DataFrame.

States = pd.DataFrame({'population': population,
'area': area

-> States.index

DataFrame can be thought as a generalization of 2-D numpy array, where both the Rows and Columns have a generalized index for accessing the data.

→ From a list of dicts. Any list of dictionaries can be made into a DataFrame.

```
Data = [{'a': i, 'b': 2*i}
        for i in range(3)]
pd.DataFrame(data)
```

Output

	a	b
0	0	0
1	1	2
2	2	4

→ From a Dictionary of Series objects.

```
pd.DataFrame({'population': population,
              'area': area})
```

→ From a 2-D Numpy Array.

Given a 2-D array of data, we can create a DataFrame with any specified column and index names. If omitted, an integer index will be used for each:

```
pd.DataFrame(np.random.rand(3,2),
             columns = ['foo', 'bar'],
             index = ['a', 'b', 'c'])
```

→ From a Numpy structured Array.

A Pandas Data Frame operates much like a structured array, and can be created directly from one.

→ A = np.zeros(3, dtype=[('A','i8'),('B','f8')])
A

Output Array([(0,0.0),(0,0.0),(0,0.0)],
dtype=[('A','<i8'),('B','<f8')])

→ pd.DataFrame(A)

The Pandas Index Object

→ This Index object is an interesting structure itself, and it can be thought of either as an immutable array or as an ordered set.

ind = pd.Index([2,3,5,7,11])

→ Print(ind.size, ind.shape, ind.ndim, ind.dtype)

→ indA = pd.Index([1,3,5,7,9])
indB = pd.Index([2,3,5,7,11])
→ indA & indB # Intersection
→ indA | indB # Union
→ indA ^ indB # Symmetric difference

→ Indexers : loc, iloc and ix

```
data = pd.Series(['a','b','c'], index=[1,3,5])
data
```

loc attribute allows indexing and slicing that always reference the explicit index.

```
data.loc[1]
```

→ 'a'

→ data.loc[1:3]

output
```
1   a
3   b
```

→ iloc attribute allows indexing and slicing that always references the implicit Python-style index.

→ data.iloc[1]

'b'

→ data.iloc[1:3]

```
3   b
5   c
```

Handling missing data

→ Numpy does provide some special aggregations that will ignore these missing values.

```
Import numpy as np
Import pandas as pd

Vals1 = np.array([1, None, 3, 4])
Vals1
```

→ NAN: Missing numerical data

```
Vals2 = np.array([1, np.nan, 3, 4])
Vals2.dtype
```

→ Everything or any operation done with Nan gives out to be NAN.

→ np.nansum(Vals2), np.nanmin(Vals2), np.nanmax(Vals2)

NAN and None in Pandas

```
pd.Series([1, np.NaN, 2, None])
x = pd.Series(range(2), dtype=int)
```

Operating on null Values

→ Pandas treats None and NaN as essentially interchangeable for indicating missing or null values.

Several useful methods are:

isnull()
Generate a Boolean mask indicating missing values

notnull()
opposite of isnull()

dropna()
Return a filtered version of the data

fillna()
Return a copy of the data with missing values filled or imputed.

Detecting null Values

→ Two useful methods for detecting null data:
is null or notnull()

→ data = pd.Series([1, np.nan, 'hello', None])
→ data.isnull()
→ data[data.notnull()]

Dropping null values.

-> dropna() [which Removes NA values].
-> fillna() (which fills in NA values).

Dropping null values in Data Frame

-> df = pd.dataframe([[1, np.nan, 2],
[2, 3, 5],
[np.nan, 4, 6]]).

-> df.

dropna() will drop all the null values present.

-> df.dropna() -> will drop all values from columns and Rows.

-> df.dropna(axis = 1 'columns'). -> dropping all null values from column it sits.

-> df[3] = np.nan.
df -> creating new column and Row of Nan value

-> df.dropna (axis = 'columns') how = 'all').

removing all the rows and columns containing NAN values.

-> df.dropna (axis = 'rows') thresh = 3).
This parameter lets you specify a min number of non-NA values for the row/column to be kept.

- data = pd.Series([1, np.nan, 2, None, 3], index = list('abcde'))
- data
- data.fillna(0) → This will fill the NAN values as zero

→ # forward-fill.
→ data.fillna(method = 'ffill')
- we can specify a forward-fill to propagate the previous values forward.

→ # back-fill
→ data.fillna(method = 'bfill')
- and we can specify a back-fill to propagate the next values backward.

→ df.fillna(method = 'ffill', axis = 1)

A multiply indexed Series

→ Representing two dimensional data within a one-d Series.

→ index = [('California', 2000), ('California', 2010),
('Newyork', 2000), ('Newyork', 2010),
('Texas', 2000), ('Texas', 2010)].

→ populations = [33871648, 37253956,
18976457, 19378102,
20851820, 25145561].

→ pop = pd.Series(populations, index = index)

→ pop

→ with this indexing scheme, you can straightly index or slice the series based on this multiple index

→ pop[('California', 2010):('Texas', 2010)].

for the values of 2010, y'all'll need to do some messy munging to make it happen.

→ pop[[i for i in pop.index if i[1] == 2010]].

Pandas multiindex

→ index = pd.MultiIndex.from_tuples(index)
→ index
→ pop = pop.reindex(index)
→ pop
→ pop[:,2010] -> ofor accessing all data from 2010.

multiIndex as extra dimension

→ pop_df = pop.unstack()
→ pop_df

unstackmethod will quickly create and convert a multiply indexed series into a conventionally indexed data frame.

→ pop_df.stack().

The stack() method provides the opposite operation.

Combining Datasets : concat and append

```
import pandas as pd
import numpy as np

def make_df(cols, ind):
    data = {c: [str(c) + str(i) for i in ind]
            for c in cols}
    return pd.DataFrame(data, ind)

make_df('ABC', range(3))
```

Simple concatenation with pd.concat

```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])

df1 = make_df('AB', [1, 2])
df2 = make_df('AB', [3, 4])
print(df1); print(df2); print(pd.concat([df1, df2]))

df3 = make_df('AB', [0, 1])
df4 = make_df('CD', [0, 1])
print(df3); print(df4); print(pd.concat([df3, df4], axis=1))
```

axis = 1.

Duplicate Indices

```
x = make_df('AB', [0, 1])
y = make_df('AB', [2, 3])
y.index = x.index ##
print(x); print(y);
print(pd.concat([x, y]))
```

Diff between

np.concatenate and pd.concat is that Pandas concatenation preserve indices, even if the result will have duplicate indices.

Ignoring the Index

-> print(x); print(y); print(pd.concat([x, y], ignore_index=True))

Adding multiIndex keys

-> keys are helpful to specify a label for the data sources the result will be a hierarchically indexed series containing the data:

-> print(pd.concat([x, y], keys=['x', 'y']))

★ Concatenation with joins

```
df5 = make-df('ABC', [1,2])
df6 = make-df('BCD', [3,4])
Print(df5); Print(df6); print(pd.concat([df5, df6]).
```

→ The join is a union of the input columns (join='outer')

outer = union

→ The inner join is the join which can be used for the intersection of the columns.

```
Print(df5); Print(df6); print(pd.concat([df5, df6],
                                join_axes=[df5.columns]))
```

★ The append method

→ Direct Array concat can be made by append method using the fewer keystrokes only.

→ For Ex:- pd.concat([df1, df2]), you can simply call df1.append(df2):

→ append method in pandas it does not modify the the original object; it creates a new object with combined data.

→ The reason is append creates a new index and data buffer

☆ Combining Datasets: merge and join

→ pd.merge() → This is a subset of what is known as Relational Algebra. which is just a formal set of rules for manipulating Relational Data.

Three categories of join

→ one-to-one join
→ many-to-one join
→ many-to-many join

(i) one-to-one join

```
df1 = pd.DataFrame({'Employee':['Bob', 'Jake', 'Lisa', 'sw'],
                    'group':['Accounting', 'Engineering', 'Engineering',
                             'HR']})
```

```
df2 = pd.DataFrame({'Employee':['Lisa', 'Bob', 'Jake', 'sw'],
                    'Hire_date':[2004, 2008, 2012, 2014]})
```

```
df3 = pd.merge(df1, df2)
```

(ii) many-to-one join

```
df4 = pd.DataFrame({'Group': ['Accounting', 'Engineering',
                              'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
print(pd.merge(df3, df4)).
pd.merge(df3, df4)
```

(iii) many-to-many joins

By performing a many-to-many join, we can recover the skills associated with any individual person:

```
df5 = pd.DataFrame({'Group': ['Accounting', 'Accounting',
                              'Engineering', 'Engineering', 'HR', 'HR'],
                    'skills': ['math', 'spreadsheets', 'coding',
                               'linux', 'spreadsheets',
                               'organization']})
```

★ df.unique() → returns array of unique values.

★ df.nunique() → gives out number of unique values

★ .corr() = shows correlation