Pandas_essentials

August 30, 2020

Pandas Essentials

====================

A short note about the essentials of the **Pandas** library that forms the basis for much of the Machine Learning Algorithms and Data Science applications in Python.

[1]:
```
import pandas as pd
```

[2]:
```
import numpy as np
```

The pandas SERIES is a one dimensional array of indexed data.

[4]:
```
data = pd.Series([0.25,0.5,0.75,1.0])
data
```

[4]:
```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

[5]:
```
data.values
```

[5]:
```
array([0.25, 0.5 , 0.75, 1.  ])
```

[6]:
```
data.index
```

[6]:
```
RangeIndex(start=0, stop=4, step=1)
```

[7]:
```
data[1]
```

[7]:
```
0.5
```

[8]:
```
data[1:3]
```

[8]:
```
1    0.50
2    0.75
dtype: float64
```

The fundamental difference between a numpy array and a pandas series is that a numpy array has implicitly defined indexes whereas a pandas series has explicitly laid out indexes. Series indexes can be manipulated to store non numeric index values.

[9]:
```
data = pd.Series([0.25,0.5,0.75,1.0],
                  index=['a','b','c','d'])
data
```

[9]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

[10]:
```
data['b']
```

[10]:
```
0.5
```

We can actually say that a pandas series is much more like a python dictionary that maps keys to values. In fact we can construct a pandas series from a dictionary as follows.

[14]:
```
population_dict = {'california':3887224,
                   'texas':223451,
                   'ny': 11256638,
                   'florida':664732,
                   'illinois':7736483}
population = pd.Series(population_dict)
population
```

[14]:
```
california     3887224
texas           223451
ny            11256638
florida         664732
illinois       7736483
dtype: int64
```

[15]:
```
population['california']
```

[15]:
```
3887224
```

[16]:
```
population['california':'illinois']
```

[16]:
```
california     3887224
texas           223451
ny            11256638
florida         664732
illinois       7736483
dtype: int64
```

Here are various ways of creating Series objects from numpy arrays and python dictionaries.

[17]:
```python
pd.Series([2,3,4])
```

[17]:
```
0    2
1    3
2    4
dtype: int64
```

[18]:
```python
pd.Series(5, index=[100,200,300])
```

[18]:
```
100    5
200    5
300    5
dtype: int64
```

[20]:
```python
pd.Series({2:'a',1:'b',3:'c'})
```

[20]:
```
2    a
1    b
3    c
dtype: object
```

Coming to the DataFrame object in pandas, we notice that it is nothing but the analog of the two dimensional numpy array and is composed of many connected Series objects.

[21]:
```python
area_dict = {'california':35638, 'texas':783674, 'new york':326733,
             'florida':11882,'illinois':2234555}
area = pd.Series(area_dict)
area
```

[21]:
```
california      35638
texas          783674
new york       326733
florida         11882
illinois      2234555
dtype: int64
```

[22]:
```python
states = pd.DataFrame({'population':population,
                       'area':area})
states
```

[22]:

	population	area
california	3887224.0	35638.0
florida	664732.0	11882.0
illinois	7736483.0	2234555.0
new york	NaN	326733.0
ny	11256638.0	NaN

```
texas          223451.0   783674.0
```

[23]:
```
states.index
```

[23]:
```
Index(['california', 'florida', 'illinois', 'new york', 'ny', 'texas'],
dtype='object')
```

[24]:
```
states.columns
```

[24]:
```
Index(['population', 'area'], dtype='object')
```

Even DataFrames have a python dictionary analog. Just like a dictionary object maps from a key to a value, the DataFrame object maps from a column name to an associated Series of column data values.

[25]:
```
states['area']
```

[25]:
```
california      35638.0
florida         11882.0
illinois      2234555.0
new york       326733.0
ny                  NaN
texas          783674.0
Name: area, dtype: float64
```

We can look at various ways of constructing dataframes.

[26]:
```
pd.DataFrame(population, columns=['population'])
```

[26]:

	population
california	3887224
texas	223451
ny	11256638
florida	664732
illinois	7736483

[29]:
```
data = [{'a':i, 'b':2*i} for i in range(3)]
pd.DataFrame(data)
```

[29]:

	a	b
0	0	0
1	1	2
2	2	4

[30]:
```
pd.DataFrame(np.random.rand(3,2),
             columns=['foo','bar'],
             index=['a','b','c'])
```

[30]:
```
        foo       bar
a  0.312593  0.287447
b  0.926330  0.628820
c  0.058090  0.505975
```

Now we look at some basic data indexing in Series.

[31]:
```python
data = pd.Series([0.25,0.5,0.75,1.0],
                 index=['a','b','c','d'])
data
```

[31]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

[32]:
```python
data['b']
```

[32]: 0.5

[33]:
```python
'a' in data
```

[33]: True

[35]:
```python
data.keys()
```

[35]: Index(['a', 'b', 'c', 'd'], dtype='object')

[36]:
```python
list(data.items())
```

[36]: [('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]

[38]:
```python
data['e'] = 0.872
data
```

[38]:
```
a    0.250
b    0.500
c    0.750
d    1.000
e    0.872
dtype: float64
```

[39]:
```python
data['a':'c']
```

[39]:
```
a    0.25
b    0.50
c    0.75
```

```
dtype: float64
```

[40]:
```python
data[0]
```

[40]:
```
0.25
```

[41]:
```python
data[0:2]
```

[41]:
```
a    0.25
b    0.50
dtype: float64
```

[42]:
```python
data[(data>0.3) & (data<0.8)]
```

[42]:
```
b    0.50
c    0.75
dtype: float64
```

[43]:
```python
data[['a','e']]
```

[43]:
```
a    0.250
e    0.872
dtype: float64
```

We know that there are implicit and explicit types of indexing in DataFrames and Series objects. Therefore to avoid confusion in indexing in the usual method, we use methods that help us index things efficiently. the LOC method is used to refer to explicit indexes.

[44]:
```python
data = pd.Series(['a','b','c'], index=[1,3,5])
data
```

[44]:
```
1    a
3    b
5    c
dtype: object
```

[45]:
```python
data[1]
```

[45]:
```
'a'
```

[46]:
```python
data[1:3]
```

[46]:
```
3    b
5    c
dtype: object
```

[47]:
```python
data.loc[1]
```

[47]: 'a'

[48]:
```
data.loc[1:3]
```

[48]:
```
1    a
3    b
dtype: object
```

Now the ILOC method allows us to index based on implicit indexing.

[49]:
```
data.iloc[1]
```

[49]: 'b'

[50]:
```
data.iloc[1:3]
```

[50]:
```
3    b
5    c
dtype: object
```

Using data selection operations in DataFrames.

[52]:
```
area = pd.Series({'california':376372, 'texas':38783, 'new york':337772,
                  'florida':115272, 'illinois':883873})
pop = pd.Series({'california':2188348, 'texas':277283, 'new york':99282,
                 'florida':7746378, 'illinois':222344})
data = pd.DataFrame({'area':area,
                     'pop':pop})
data
```

[52]:

	area	pop
california	376372	2188348
texas	38783	277283
new york	337772	99282
florida	115272	7746378
illinois	883873	222344

[53]:
```
data['area']
```

[53]:
```
california    376372
texas          38783
new york      337772
florida       115272
illinois      883873
Name: area, dtype: int64
```

[54]:
```
data.area
```

[54]:
```
california    376372
texas          38783
new york      337772
florida       115272
illinois      883873
Name: area, dtype: int64
```

We can add a ne column by using vectorized operations of the other columns.

[55]:
```
data['density'] = data['pop']/data['area']
data
```

[55]:
```
              area      pop   density
california  376372  2188348  5.814322
texas        38783   277283  7.149602
new york    337772    99282  0.293932
florida     115272  7746378 67.200864
illinois    883873   222344  0.251557
```

[59]:
```
data.values
```

[59]:
```
array([[3.76372000e+05, 2.18834800e+06, 5.81432200e+00],
       [3.87830000e+04, 2.77283000e+05, 7.14960163e+00],
       [3.37772000e+05, 9.92820000e+04, 2.93932001e-01],
       [1.15272000e+05, 7.74637800e+06, 6.72008640e+01],
       [8.83873000e+05, 2.22344000e+05, 2.51556502e-01]])
```

Since the DataFrame is essentially a special kind of two dimensional array or matrix, we can perform array computations on it like transposing.

[60]:
```
data.T
```

[60]:
```
           california          texas       new york       florida  \
area     3.763720e+05   38783.000000  337772.000000  1.152720e+05
pop      2.188348e+06  277283.000000   99282.000000  7.746378e+06
density  5.814322e+00       7.149602       0.293932  6.720086e+01

              illinois
area     883873.000000
pop      222344.000000
density       0.251557
```

Since the values of a dataframe is essentially a numpy array, we can index it like a numpy array.

[62]:
```
data.values[2]
```

[62]:
```
array([3.37772000e+05, 9.92820000e+04, 2.93932001e-01])
```

[63]:
```
data['area']
```

[63]:
```
california    376372
texas          38783
new york      337772
florida       115272
illinois      883873
Name: area, dtype: int64
```

We can use iloc and loc indexers to index using the index first and column second notations.

[66]:
```
data.iloc[:3,:2]
```

[66]:
```
              area      pop
california  376372  2188348
texas        38783   277283
new york    337772    99282
```

[68]:
```
data.loc[:'new york',:'pop']
```

[68]:
```
              area      pop
california  376372  2188348
texas        38783   277283
new york    337772    99282
```

[70]:
```
data.loc[:,:'density']
```

[70]:
```
              area      pop    density
california  376372  2188348   5.814322
texas        38783   277283   7.149602
new york    337772    99282   0.293932
florida     115272  7746378  67.200864
illinois    883873   222344   0.251557
```

[71]:
```
data.loc['texas':'florida', 'pop':'density']
```

[71]:
```
              pop    density
texas      277283   7.149602
new york    99282   0.293932
florida   7746378  67.200864
```

[75]:
```
data.loc[data.density>2, 'pop':'density']
```

[75]:
```
                pop    density
california  2188348   5.814322
texas        277283   7.149602
florida     7746378  67.200864
```

[76]:
```
data.loc[data['density']>2, ['area','density']]
```

[76]:

	area	density
california	376372	5.814322
texas	38783	7.149602
florida	115272	67.200864

[77]:
```
data.iloc[0,2] = 90
data
```

[77]:

	area	pop	density
california	376372	2188348	90.000000
texas	38783	277283	7.149602
new york	337772	99282	0.293932
florida	115272	7746378	67.200864
illinois	883873	222344	0.251557

[78]:
```
data['florida':'illinois']
```

[78]:

	area	pop	density
florida	115272	7746378	67.200864
illinois	883873	222344	0.251557

[79]:
```
data[1:3]
```

[79]:

	area	pop	density
texas	38783	277283	7.149602
new york	337772	99282	0.293932

[80]:
```
data[data.density>2]
```

[80]:

	area	pop	density
california	376372	2188348	90.000000
texas	38783	277283	7.149602
florida	115272	7746378	67.200864

Where there is missing data, Python pandas has a special reserved floating point value to represent that missing value and is denoted as NaN.

[81]:
```
vals2 = np.array([1,np.nan,3,4])
vals2.dtype
```

[81]:
```
dtype('float64')
```

Typically, general aggregating functions used across DataFrames and arrays are usually spoilt due to the nan value and hence we must use other functions.

[84]:
```
vals2.sum(), vals2.min(), vals2.max()
```

[84]: (nan, nan, nan)

The following are the corresponding aggregation functions that ignore nan values in computation.

[85]:
```
np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
```

[85]: (8.0, 1.0, 4.0)

We can use certain functions that detects the null and nonnull values in the data. They ouput a boolean mask over the data elements.

[86]:
```
data = pd.Series([1, np.nan, 'hello', None])
data.isnull()
```

[86]:
```
0    False
1     True
2    False
3     True
dtype: bool
```

[87]:
```
data[data.notnull()]
```

[87]:
```
0        1
2    hello
dtype: object
```

Removing null values.

[88]:
```
data.dropna()
```

[88]:
```
0        1
2    hello
dtype: object
```

When it comes to dropping values in a multidimensional dataframe, we cannot remove singled out nan values, rather we drop the row or column corresponding to a nan value.

[90]:
```
df = pd.DataFrame([[1, np.nan, 2],
                  [2, 3, 5],
                  [np.nan, 4, 6]])
df
```

[90]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

[91]:
```
df.dropna()
```

[91]:
```
     0    1  2
1  2.0  3.0  5
```

[92]:
```
df.dropna(axis='columns')
```

[92]:
```
   2
0  2
1  5
2  6
```

We can use the fillna() method to replace nan values with some integer or floating point value.

[93]:
```
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
data
```

[93]:
```
a    1.0
b    NaN
c    2.0
d    NaN
e    3.0
dtype: float64
```

[94]:
```
data.fillna(0)
```

[94]:
```
a    1.0
b    0.0
c    2.0
d    0.0
e    3.0
dtype: float64
```

We can fill nan values with the previous legitimate value using a method known as forward fill.

[96]:
```
data.fillna(method='ffill')
```

[96]:
```
a    1.0
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64
```

Using back fill to fill nan values.

[97]:
```
data.fillna(method='bfill')
```

[97]:
```
a    1.0
b    2.0
```

```
c    2.0
d    3.0
e    3.0
dtype: float64
```

[99]:
```
df.fillna(method='ffill', axis=1)
```

[99]:

	0	1	2
0	1.0	1.0	2.0
1	2.0	3.0	5.0
2	NaN	4.0	6.0

Concatenating dataframes and series using the concat() function.

[105]:
```
ser1 = pd.Series(['A','B','C'], index=[1,2,3])
ser2 = pd.Series(['D','E','F'], index=[3,5,6])
pd.concat([ser1, ser2])
```

[105]:
```
1    A
2    B
3    C
3    D
5    E
6    F
dtype: object
```

Sometimes when we are concatenating two dataframes or series with overlapping index values, the index values are simply repeated in the concatenated dataframe. If we want to verify the presence of overlapping indexes, we use the verify_integrity flag and raise an exception if duplicate indexes come out.

[106]:
```
try:
    pd.concat([ser1, ser2], verify_integrity=True)
except ValueError as e:
    print("Value error: ", e)
```

```
Value error:  Indexes have overlapping values: Int64Index([3], dtype='int64')
```

If we are forming a new dataframe by concatenating two other dataframes, we can choose to ignore the index of the original ones and create a new index.

[107]:
```
print(pd.concat([ser1, ser2], ignore_index=True))
```

```
0    A
1    B
2    C
3    D
4    E
```

```
5    F
dtype: object
```

Before turning our attention to concatenating dataframes using joins, here is a small function that custom creates a dataframe using a simple rule.

[108]:
```
def make_df(cols, ind):
    data = {c: [str(c) + str(i) for i in ind] for c in cols}
    return pd.DataFrame(data, ind)
```

[110]:
```
df5 = make_df('ABC', [1,2])
df6 = make_df('BCD', [3,4])
print(df5), print(df6), print(pd.concat([df5, df6]))
```

```
    A   B   C
1  A1  B1  C1
2  A2  B2  C2
    B   C   D
3  B3  C3  D3
4  B4  C4  D4
     A   B   C    D
1   A1  B1  C1  NaN
2   A2  B2  C2  NaN
3  NaN  B3  C3   D3
4  NaN  B4  C4   D4

/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
FutureWarning: Sorting because non-concatenation axis 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'.

  This is separate from the ipykernel package so we can avoid doing imports
until
```

[110]:
```
(None, None, None)
```

By defualt, we notice that the entries for which data is not available is filled with NA values. Use inner join to fetch only those columns which are common among both the concatenated dataframes.

[111]:
```
print(pd.concat([df5, df6], join='inner'))
```

```
    B   C
1  B1  C1
2  B2  C2
3  B3  C3
```

```
4  B4  C4
```

We can use the join_axes() argument which essentially fetches the indexes of columns according to which we want to join the data. Here we join the data according to the first dataframes columns.

[118]:
```python
print(pd.concat([df5, df6], join_axes=[df5.columns]))
```

```
     A   B   C
1   A1  B1  C1
2   A2  B2  C2
3  NaN  B3  C3
4  NaN  B4  C4

/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:1:
FutureWarning: The join_axes-keyword is deprecated. Use .reindex or
.reindex_like on the result to achieve the same functionality.
  """Entry point for launching an IPython kernel.
```

We can also use the append() method to concatenate two dataframes.

[115]:
```python
print(df5.append(df6))
```

```
     A   B   C    D
1   A1  B1  C1  NaN
2   A2  B2  C2  NaN
3  NaN  B3  C3   D3
4  NaN  B4  C4   D4
```

We now explore the one-to-one join which is essentially a column wise concatenation of dataframes. The Merge function recognizes the presence of a common column in the two dataframes and joins the dataframes using this column as the key.

[119]:
```python
df1 = pd.DataFrame({'employee':['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group':['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee':['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date':[2004, 2008, 2012, 2014]})
print(df1)
print(df2)
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2     Jake       2012
3      Sue       2014
```

[122]:
```
df3 = pd.merge(df1, df2)
df3
```

[122]:

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

In a many-to-one join, two key-columns might have duplicate entries. This join will preserve those duplicate entries as it is found appropriate.

[123]:
```
df4 = pd.DataFrame({'group':['Accounting', 'Engineering', 'HR'],
                    'supervisor':['Carly', 'Guido', 'Steve']})
print(df3)
print(df4)
print(pd.merge(df3,df4))
```

```
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
         group supervisor
0   Accounting      Carly
1  Engineering      Guido
2           HR      Steve
  employee        group  hire_date supervisor
0      Bob   Accounting       2008      Carly
1     Jake  Engineering       2012      Guido
2     Lisa  Engineering       2004      Guido
3      Sue           HR       2014      Steve
```

Finally, many-to-many joins happen when both dataframes have key columns with duplicate values.

[125]:
```
df5 = pd.DataFrame({'group':['Accounting', 'Accounting', 'Engineering',
                             'Engineering', 'HR', 'HR'],
                    'skills':['math', 'spreadsheets', 'coding', 'linus',
                              'spreadsheets', 'organization']})
print(df1)
print(df5)
print(pd.merge(df1, df5))
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
         group        skills
```

```
0   Accounting         math
1   Accounting  spreadsheets
2  Engineering        coding
3  Engineering         linus
4           HR  spreadsheets
5           HR  organization
  employee        group        skills
0      Bob   Accounting          math
1      Bob   Accounting  spreadsheets
2     Jake  Engineering        coding
3     Jake  Engineering         linus
4     Lisa  Engineering        coding
5     Lisa  Engineering         linus
6      Sue           HR  spreadsheets
7      Sue           HR  organization
```

We can also explicitly specify the name of the key column on the basis of which we wish to conduct a join or concatenation operation on various dataframes.

[126]:
```
print(df1)
print(df2)
print(pd.merge(df1, df2, on='employee'))
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2     Jake       2012
3      Sue       2014
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
```

In situations where the supposed key columns of two dataframes contain similar meaningful values but have different column names we use the left_on and right_on arguments.

[127]:
```
df3 = pd.DataFrame({'name':['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary':[70000, 80000, 120000, 90000]})
print(df1)
print(df3)
print(pd.merge(df1, df3, left_on='employee', right_on='name'))
```

```
  employee        group
```

```
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
   name  salary
0   Bob   70000
1  Jake   80000
2  Lisa  120000
3   Sue   90000
  employee        group  name  salary
0      Bob   Accounting   Bob   70000
1     Jake  Engineering  Jake   80000
2     Lisa  Engineering  Lisa  120000
3      Sue           HR   Sue   90000
```

In the above joined dataframe we ended up with a redundant column name which we can remove by specifying the drop argument.

[131]:
```
pd.merge(df1, df3, left_on='employee', right_on='name').drop('name', axis=1)
```

[131]:
```
  employee        group  salary
0      Bob   Accounting   70000
1     Jake  Engineering   80000
2     Lisa  Engineering  120000
3      Sue           HR   90000
```

We can use the set_index() function to set a specific column as an index to the dataframe and then join on the basis of that index.

[132]:
```
df1a = df1.set_index('employee')
df2a = df2.set_index('employee')
print(df1a)
print(df2a)
print(pd.merge(df1a, df2a, left_index=True, right_index=True))
```

```
                group
employee
Bob        Accounting
Jake      Engineering
Lisa      Engineering
Sue                HR
          hire_date
employee
Lisa           2004
Bob            2008
Jake           2012
Sue            2014
                group  hire_date
employee
```

```
Bob        Accounting       2008
Jake      Engineering       2012
Lisa      Engineering       2004
Sue                HR       2014
```

The same operation can be performed using join() which by default uses the index as the basis for joining.

[133]:
```python
print(df1a.join(df2a))
```

```
                group  hire_date
employee
Bob        Accounting       2008
Jake      Engineering       2012
Lisa      Engineering       2004
Sue                HR       2014
```

We can also mix and match and join dataframes as per index and a specific column.

[135]:
```python
print(df1a)
print(df3)
print(pd.merge(df1a, df3, left_index=True, right_on='name'))
```

```
                group
employee
Bob        Accounting
Jake      Engineering
Lisa      Engineering
Sue                HR
   name  salary
0   Bob   70000
1  Jake   80000
2  Lisa  120000
3   Sue   90000
         group  name  salary
0   Accounting   Bob   70000
1  Engineering  Jake   80000
2  Engineering  Lisa  120000
3           HR   Sue   90000
```

We can use set arithmetic to perform more dynamic joins, that is when the common key column in different dataframes do not contain the same elements. First we see the inner join which the intersection of two sets.

[4]:
```python
df6 = pd.DataFrame({'name':['Peter', 'Paul', 'Mary'],
                    'food':['fish', 'beans', 'bread']},
                   columns=['name', 'food'])
df7 = pd.DataFrame({'name':['Mary', 'Joseph'],
                    'drink':['wine', 'beer']},
                   columns=['name', 'drink'])
```

```python
print(df6)
print(df7)
print(pd.merge(df6, df7, how='inner'))
```

```
    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread
     name drink
0    Mary  wine
1  Joseph  beer
   name   food drink
0  Mary  bread  wine
```

Outer join a union of input column elements wherein all the missing values are filled with nan values.

[137]:
```python
print(df6)
print(df7)
print(pd.merge(df6, df7, how='outer'))
```

```
    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread
     name drink
0    Mary  wine
1  Joseph  beer
     name   food drink
0   Peter   fish   NaN
1    Paul  beans   NaN
2    Mary  bread  wine
3  Joseph    NaN  beer
```

Similarly, left and right joins end up joining the dataframes based on the left or right dataframe column values.

[5]:
```python
print(df6)
print(df7)
print(pd.merge(df6, df7, how='left'))
```

```
    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread
     name drink
0    Mary  wine
1  Joseph  beer
    name   food drink
```

```
0  Peter   fish   NaN
1   Paul  beans   NaN
2   Mary  bread  wine
```

[6]:
```
import seaborn as sns
planets = sns.load_dataset('planets')
planets.shape
```

[6]:
```
(1035, 6)
```

[7]:
```
planets.head()
```

[7]:
```
            method  number  orbital_period   mass  distance  year
0  Radial Velocity       1         269.300   7.10     77.40  2006
1  Radial Velocity       1         874.774   2.21     56.95  2008
2  Radial Velocity       1         763.000   2.60     19.84  2011
3  Radial Velocity       1         326.030  19.40    110.62  2007
4  Radial Velocity       1         516.220  10.50    119.47  2009
```

[11]:
```
rng = np.random.RandomState(42)
ser = pd.Series(rng.rand(5))
ser
```

[11]:
```
0    0.374540
1    0.950714
2    0.731994
3    0.598658
4    0.156019
dtype: float64
```

[12]:
```
ser.sum()
```

[12]:
```
2.811925491708157
```

[13]:
```
ser.mean()
```

[13]:
```
0.5623850983416314
```

[14]:
```
df = pd.DataFrame({'A': rng.rand(5),
                   'B': rng.rand(5)})
df
```

[14]:
```
          A         B
0  0.155995  0.020584
1  0.058084  0.969910
2  0.866176  0.832443
3  0.601115  0.212339
```

```
4  0.708073  0.181825
```

[15]:
```
df.mean()
```

[15]:
```
A    0.477888
B    0.443420
dtype: float64
```

[16]:
```
df.mean(axis='columns')
```

[16]:
```
0    0.088290
1    0.513997
2    0.849309
3    0.406727
4    0.444949
dtype: float64
```

[17]:
```
planets.dropna().describe()
```

[17]:

	number	orbital_period	mass	distance	year
count	498.00000	498.000000	498.000000	498.000000	498.000000
mean	1.73494	835.778671	2.509320	52.068213	2007.377510
std	1.17572	1469.128259	3.636274	46.596041	4.167284
min	1.00000	1.328300	0.003600	1.350000	1989.000000
25%	1.00000	38.272250	0.212500	24.497500	2005.000000
50%	1.00000	357.000000	1.245000	39.940000	2009.000000
75%	2.00000	999.600000	2.867500	59.332500	2011.000000
max	6.00000	17337.500000	25.000000	354.000000	2014.000000

Using the groupby() method to split, apply and combine. This essentially means that we first split the dataset as per some groupings, then apply some form of aggregation or condition on the split datasets and finally combine them to form grouped aggregates.

[18]:
```
df = pd.DataFrame({'key': ['A','B','C','A','B','C'],
                   'data': range(6)}, columns=['key', 'data'])
df
```

[18]:

	key	data
0	A	0
1	B	1
2	C	2
3	A	3
4	B	4
5	C	5

[19]:
```
df.groupby('key')
```

[19]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x1a1fb65c10>
```

[20]:
```
df.groupby('key').sum()
```

[20]:
```
     data
key
A       3
B       5
C       7
```

Column indexing using groupby() method.

[21]:
```
planets.groupby('method')
```

[21]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x1a1fb54b50>
```

[23]:
```
planets.groupby('method')['orbital_period']
```

[23]:
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x1a2055bf50>
```

[24]:
```
planets.groupby('method')['orbital_period'].median()
```

[24]:
```
method
Astrometry                         631.180000
Eclipse Timing Variations         4343.500000
Imaging                          27500.000000
Microlensing                      3300.000000
Orbital Brightness Modulation        0.342887
Pulsar Timing                       66.541900
Pulsation Timing Variations       1170.000000
Radial Velocity                    360.200000
Transit                              5.714932
Transit Timing Variations           57.011000
Name: orbital_period, dtype: float64
```

[28]:
```
planets.groupby('method')['year'].describe().unstack()
```

[28]:
```
       method
count  Astrometry                          2.0
       Eclipse Timing Variations           9.0
       Imaging                            38.0
       Microlensing                       23.0
       Orbital Brightness Modulation       3.0
                                        …
max    Pulsar Timing                    2011.0
       Pulsation Timing Variations      2007.0
       Radial Velocity                  2014.0
```

```
        Transit                              2014.0
        Transit Timing Variations            2014.0
Length: 80, dtype: float64
```

Using the filter, apply, tranform and aggregate functionality.

[29]:
```
rng = np.random.RandomState(0)
df = pd.DataFrame({'key': ['A','B','C','A','B','C'],
                   'data1': range(6),
                   'data2': rng.randint(0, 10, 6)},
                  columns=['key', 'data1', 'data2'])
df
```

[29]:
```
  key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9
```

[31]:
```
df.groupby('key').describe().unstack()
```

[31]:
```
              key
data1  count  A      2.000000
              B      2.000000
              C      2.000000
       mean   A      1.500000
              B      2.500000
              C      3.500000
       std    A      2.121320
              B      2.121320
              C      2.121320
       min    A      0.000000
              B      1.000000
              C      2.000000
       25%    A      0.750000
              B      1.750000
              C      2.750000
       50%    A      1.500000
              B      2.500000
              C      3.500000
       75%    A      2.250000
              B      3.250000
              C      4.250000
       max    A      3.000000
              B      4.000000
```

```
              C       5.000000
data2  count  A       2.000000
              B       2.000000
              C       2.000000
       mean   A       4.000000
              B       3.500000
              C       6.000000
       std    A       1.414214
              B       4.949747
              C       4.242641
       min    A       3.000000
              B       0.000000
              C       3.000000
       25%    A       3.500000
              B       1.750000
              C       4.500000
       50%    A       4.000000
              B       3.500000
              C       6.000000
       75%    A       4.500000
              B       5.250000
              C       7.500000
       max    A       5.000000
              B       7.000000
              C       9.000000
dtype: float64
```

[35]:
```
df.groupby('key').aggregate(['min', np.median, max])
```

[35]:
```
    data1            data2
      min median max   min median max
key
A       0    1.5   3     3    4.0   5
B       1    2.5   4     0    3.5   7
C       2    3.5   5     3    6.0   9
```

[34]:
```
df.groupby('key').aggregate({'data1': 'min',
                             'data2': 'max'})
```

[34]:
```
     data1  data2
key
A        0      5
B        1      7
C        2      9
```

The filter operation lets us drop rows with certain group properties.

[37]:
```
def filter_func(x):
    return x['data2'].std() > 4

print(df)
print(df.groupby('key').std())
print(df.groupby('key').filter(filter_func))
```

```
  key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9
       data1     data2
key
A    2.12132  1.414214
B    2.12132  4.949747
C    2.12132  4.242641
  key  data1  data2
1   B      1      0
2   C      2      3
4   B      4      7
5   C      5      9
```

Using the transform operation to transform a data matrix such as centering its values. Here we center using the group wise mean.

[38]:
```
df.groupby('key').transform(lambda x: x - x.mean())
```

[38]:

	data1	data2
0	-1.5	1.0
1	-1.5	-3.5
2	-1.5	-3.0
3	1.5	-1.0
4	1.5	3.5
5	1.5	3.0

We can use the apply() function to apply any function to the data. Here we will normalize the first column with the sum of the second column. Note that the operation is done group wise.

[40]:
```
def norm_by_data2(x):
    x['data1'] /= x['data2'].sum()
    return x

print(df)
print(df.groupby('key').apply(norm_by_data2))
```

```
  key  data1  data2
```

```
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9
  key     data1  data2
0   A  0.000000      5
1   B  0.142857      0
2   C  0.166667      3
3   A  0.375000      3
4   B  0.571429      7
5   C  0.416667      9
```

We can actually specify such grouping keys as custom defined lists or arrays also. They don't always have to be column names.

[41]:
```
L = [0,1,0,1,2,0]
print(df)
print(df.groupby(L).sum())
```

```
  key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9
   data1  data2
0      7     17
1      4      3
2      4      7
```

[42]:
```
print(df)
print(df.groupby('key').sum())
```

```
  key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9
     data1  data2
key
A        3      8
B        5      7
C        7     12
```

As another key specification we can create a dictionary that maps index values to key values.

[44]:
```
df2 = df.set_index('key')
mapping = {'A': 'vowel', 'B': 'consonant', 'C': 'consonant'}
print(df2)
print(df2.groupby(mapping).sum())
```

```
     data1  data2
key
A        0      5
B        1      0
C        2      3
A        3      3
B        4      7
C        5      9
           data1  data2
consonant     12     19
vowel          3      8
```

[45]:
```
print(df2)
print(df2.groupby(str.lower).sum())
```

```
     data1  data2
key
A        0      5
B        1      0
C        2      3
A        3      3
B        4      7
C        5      9
   data1  data2
a      3      8
b      5      7
c      7     12
```

[46]:
```
print(df2.groupby([str.lower, mapping]).sum())
```

```
             data1  data2
a vowel          3      8
b consonant      5      7
c consonant      7     12
```

[56]:
```
decade = 10 * (planets['year'] // 10)
decade = decade.astype(str) + 's'
decade.name = 'decade'
planets.groupby(['method', decade])['number'].sum().unstack().fillna(0)
```

[56]:
```
decade                         1980s  1990s  2000s  2010s
method
Astrometry                       0.0    0.0    0.0    2.0
Eclipse Timing Variations        0.0    0.0    5.0   10.0
Imaging                          0.0    0.0   29.0   21.0
Microlensing                     0.0    0.0   12.0   15.0
Orbital Brightness Modulation    0.0    0.0    0.0    5.0
Pulsar Timing                    0.0    9.0    1.0    1.0
Pulsation Timing Variations      0.0    0.0    1.0    0.0
Radial Velocity                  1.0   52.0  475.0  424.0
Transit                          0.0    0.0   64.0  712.0
Transit Timing Variations        0.0    0.0    0.0    9.0
```

A Pivot table is a similar operation like a groupby. The pivot table essentially takes column wise data as inputs, groups data and then provides a multidimensional summarization of that data. Pivots are essentially multidimensional aggregations of groupbys.

[57]:
```python
titanic = sns.load_dataset('titanic')
titanic.head()
```

[57]:
```
   survived  pclass     sex   age  sibsp  parch     fare embarked  class  \
0         0       3    male  22.0      1      0   7.2500        S  Third
1         1       1  female  38.0      1      0  71.2833        C  First
2         1       3  female  26.0      0      0   7.9250        S  Third
3         1       1  female  35.0      1      0  53.1000        S  First
4         0       3    male  35.0      0      0   8.0500        S  Third

     who  adult_male deck  embark_town alive  alone
0    man        True  NaN  Southampton    no  False
1  woman       False    C    Cherbourg   yes  False
2  woman       False  NaN  Southampton   yes   True
3  woman       False    C  Southampton   yes  False
4    man        True  NaN  Southampton    no   True
```

[59]:
```python
titanic.groupby('sex')[['survived']].mean()
```

[59]:
```
        survived
sex
female  0.742038
male    0.188908
```

Now in the following we will basically do the following: group by class and gender, select survival, apply a mean aggregate, combine the resulting groups and finally unstack the hierarchical index to obtain a cleaner dimensionality in the data.

[60]:
```python
titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').unstack()
```

[60]:
```
class      First    Second     Third
sex
female  0.968085  0.921053  0.500000
male    0.368852  0.157407  0.135447
```

We can do the above same operation using the pivot_table method.

[62]:
```
titanic.pivot_table('survived', index='sex', columns='class')
```

[62]:
```
class      First    Second     Third
sex
female  0.968085  0.921053  0.500000
male    0.368852  0.157407  0.135447
```

References - Python DataScience Handbook