Practical Tutorial on Data Manipulation with Numpy and Pandas in Python

The pandas library has emerged into a power house of data manipulation tasks in python since it was developed in 2008. With its intuitive syntax and flexible data structure, it's easy to learn and enables faster data computation. The development of numpy and pandas libraries has extended python's multi-purpose nature to solve machine learning problems as well. The acceptance of python language in machine learning has been phenomenal since then.

Note: This tutorial is best suited for people who know the basics of python. No further knowledge is expected. Make sure you have python installed on your laptop.

Table of Contents

6 Important things you should know about Numpy and Pandas

1. The data manipulation capabilities of pandas are built on top of the numpy library. In a way, numpy is a dependency of the pandas library.
2. Pandas is best at handling tabular data sets comprising different variable types (integer, float, double, etc.). In addition, the pandas library can also be used to perform even the most naive of tasks such as loading data or doing feature engineering on time series data.
3. Numpy is most suitable for performing basic numerical computations such as mean, median, range, etc. Alongside, it also supports the creation of multi-dimensional arrays.
4. Numpy library can also be used to integrate C/C++ and Fortran code.
5. Remember, python is a zero indexing language unlike R where indexing starts at one.
6. The best part of learning pandas and numpy is the strong active community support you'll get from around the world.

Just to give you a flavor of the numpy library, we'll quickly go through its syntax structures and some important commands such as slicing, indexing, concatenation, etc. All these commands will come in handy when using pandas as well. Let's get started!

Starting with Numpy

```
#load the library and check its version, just to make sure we aren't using an older version
import numpy as np
np.__version__
'1.12.1'
#create a list comprising numbers from 0 to 9
L = list(range(10))
#converting integers to string - this style of handling lists is known as list comprehension.
#List comprehension offers a versatile way to handle list manipulations tasks easily. We'll learn about them in future tutorials. Here's an example.

[str(c) for c in L]
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

[type(item) for item in L]
[int, int, int, int, int, int, int, int, int, int]
```

Creating Arrays

Numpy arrays are homogeneous in nature, i.e., they comprise one data type (integer, float, double, etc.) unlike lists.

```
#creating arrays
np.zeros(10, dtype='int')
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])


#creating a 3 row x 5 column matrix
np.ones((3,5), dtype=float)
array([[ 1.,  1.,  1.,  1.,  1.],
      [ 1.,  1.,  1.,  1.,  1.],
      [ 1.,  1.,  1.,  1.,  1.]])


#creating a matrix with a predefined value
np.full((3,5),1.23)
array([[ 1.23,  1.23,  1.23,  1.23,  1.23],
      [ 1.23,  1.23,  1.23,  1.23,  1.23],
      [ 1.23,  1.23,  1.23,  1.23,  1.23]])


#create an array with a set sequence
np.arange(0, 20, 2)
array([0, 2, 4, 6, 8,10,12,14,16,18])


#create an array of even space between the given range of values
np.linspace(0, 1, 5)
array([ 0., 0.25, 0.5 , 0.75, 1.])


#create a 3x3 array with mean 0 and standard deviation 1 in a given dimension
np.random.normal(0, 1, (3,3))
array([[ 0.72432142, -0.90024075,  0.27363808],
      [ 0.88426129,  1.45096856, -1.03547109],
      [-0.42930994, -1.02284441, -1.59753603]])


#create an identity matrix
np.eye(3)
array([[ 1.,  0.,  0.],
      [ 0.,  1.,  0.],
      [ 0.,  0.,  1.]])


#set a random seed
np.random.seed(0)


x1 = np.random.randint(10, size=6) #one dimension
x2 = np.random.randint(10, size=(3,4)) #two dimension
x3 = np.random.randint(10, size=(3,4,5)) #three dimension
```

```
print("x3 ndim:", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
('x3 ndim:', 3)
('x3 shape:', (3, 4, 5))
('x3 size: ', 60)
```

Array Indexing

The important thing to remember is that indexing in python starts at zero.

```
x1 = np.array([4, 3, 4, 4, 8, 4])
x1
array([4, 3, 4, 4, 8, 4])

#assess value to index zero
x1[0]
4

#assess fifth value
x1[4]
8

#get the last value
x1[-1]
4

#get the second last value
x1[-2]
8

#in a multidimensional array, we need to specify row and column index
x2
array([[3, 7, 5, 5],
       [0, 1, 5, 9],
       [3, 0, 5, 0]])


#1st row and 2nd column value
x2[2,3]
0

#3rd row and last value from the 3rd column
x2[2,-1]
0


#replace value at 0,0 index
x2[0,0] = 12
x2
array([[12,  7,  5,  5],
       [ 0,  1,  5,  9],
      [ 3,  0,  5,  0]])
```

Array Slicing

Now, we'll learn to access multiple or a range of elements from an array.

```
x = np.arange(10)
x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])


#from start to 4th position
x[:5]
array([0, 1, 2, 3, 4])


#from 4th position to end
x[4:]
array([4, 5, 6, 7, 8, 9])


#from 4th to 6th position
x[4:7]
array([4, 5, 6])


#return elements at even place
x[ : : 2]
array([0, 2, 4, 6, 8])


#return elements from first position step by two
x[1::2]
array([1, 3, 5, 7, 9])


#reverse the array
x[::-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

Array Concatenation

Many a time, we are required to combine different arrays. So, instead of typing each of their elements manually, you can use array concatenation to handle such tasks easily.

```
#You can concatenate two or more arrays at once.
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
z = [21,21,21]
np.concatenate([x, y,z])
array([ 1,  2,  3,  3,  2,  1, 21, 21, 21])
```

```
#You can also use this function to create 2-dimensional arrays.
grid = np.array([[1,2,3],[4,5,6]])
np.concatenate([grid,grid])
array([[1, 2, 3],
      [4, 5, 6],
      [1, 2, 3],
      [4, 5, 6]])


#Using its axis parameter, you can define row-wise or column-wise matrix
np.concatenate([grid,grid],axis=1)
array([[1, 2, 3, 1, 2, 3],
      [4, 5, 6, 4, 5, 6]])
```

Until now, we used the concatenation function of arrays of equal dimension. But, what if you are required to combine a 2D array with 1D array? In such situations, np.concatenate might not be the best option to use. Instead, you can use np.vstack or np.hstack to do the task. Let's see how!

```
x = np.array([3,4,5])
grid = np.array([[1,2,3],[17,18,19]])
np.vstack([x,grid])
array([[ 3,  4,  5],
      [ 1,  2,  3],
      [17, 18, 19]])


#Similarly, you can add an array using np.hstack
z = np.array([[9],[9]])
np.hstack([grid,z])
array([[ 1,  2,  3,  9],
      [17, 18, 19,  9]])
```

Also, we can split the arrays based on pre-defined positions. Let's see how!

```
x = np.arange(10)
x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])


x1,x2,x3 = np.split(x,[3,6])
print x1,x2,x3
[0 1 2] [3 4 5] [6 7 8 9]

grid = np.arange(16).reshape((4,4))
grid
upper,lower = np.vsplit(grid,[2])
print (upper, lower)
(array([[0, 1, 2, 3],
       [4, 5, 6, 7]]), array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]]))
```

In addition to the functions we learned above, there are several other mathematical functions available in the numpy library such as sum, divide, multiple, abs, power, mod, sin, cos, tan, log, var, min, mean, max, etc. which you can be used to perform basic arithmetic calculations. Feel free to refer to numpy documentation for more information on such functions.

Let's move on to pandas now. Make sure you following each line below because it'll help you in doing data manipulation using pandas.

Let's start with Pandas

```
#load library - pd is just an alias. I used pd because it's short and literally abbreviates pandas.
#You can use any name as an alias.
import pandas as pd
#create a data frame - dictionary is used here where keys get converted to column names and values to row values.
data = pd.DataFrame({'Country': ['Russia','Colombia','Chile','Equador','Nigeria'],
                    'Rank':[121,40,100,130,11]})
data
```

	Country	Rank
0	Russia	121
1	Colombia	40
2	Chile	100
3	Equador	130
4	Nigeria	11

```
#We can do a quick analysis of any data set using:
data.describe()
```

	Rank
count	5.000000

	Rank
mean	80.400000
std	52.300096
min	11.000000
25%	40.000000
50%	100.000000
75%	121.000000
max	130.000000

Remember, describe() method computes summary statistics of integer / double variables. To get the complete information about the data set, we can use info() function.

```
#Among other things, it shows the data set has 5 rows and 2 columns with their
respective names.
data.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 2 columns):
Country    5 non-null object
Rank       5 non-null int64
dtypes: int64(1), object(1)
memory usage: 152.0+ bytes


#Let's create another data frame.
data  =  pd.DataFrame({'group':['a',  'a',  'a',  'b','b',  'b',  'c',
'c','c'],'ounces':[4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
data
```

	group	ounces
0	a	4.0
1	a	3.0
2	a	12.0
3	b	6.0
4	b	7.5
5	b	8.0
6	c	3.0
7	c	5.0
8	c	6.0

```
#Let's sort the data frame by ounces - inplace = True will make changes to the
data
data.sort_values(by=['ounces'],ascending=True,inplace=False)
```

	group	ounces
1	a	3.0

	group	ounces
6	c	3.0
0	a	4.0
7	c	5.0
3	b	6.0
8	c	6.0
4	b	7.5
5	b	8.0
2	a	12.0

We can sort the data by not just one column but multiple columns as well.

```
data.sort_values(by=['group','ounces'],ascending=[True,False],inplace=False)
```

	group	ounces
2	a	12.0
0	a	4.0

	group	ounces
1	a	3.0
5	b	8.0
4	b	7.5
3	b	6.0
8	c	6.0
7	c	5.0
6	c	3.0

Often, we get data sets with duplicate rows, which is nothing but noise. Therefore, before training the model, we need to make sure we get rid of such inconsistencies in the data set. Let's see how we can remove duplicate rows.

```
#create another data with duplicated rows
data = pd.DataFrame({'k1':['one']*3 + ['two']*4, 'k2':[3,2,1,3,3,4,4]})
data
```

	k1	k2
0	one	3
1	one	2

	k1	**k2**
2	one	1
3	two	3
4	two	3
5	two	4
6	two	4

```
#sort values
data.sort_values(by='k2')
```

	k1	**k2**
2	one	1
1	one	2
0	one	3
3	two	3
4	two	3
5	two	4

	k1	k2
6	two	4

```
#remove duplicates - ta da!
data.drop_duplicates()
```

	k1	k2
0	one	3
1	one	2
2	one	1
3	two	3
5	two	4

Here, we removed duplicates based on matching row values across all columns. Alternatively, we can also remove duplicates based on a particular column. Let's remove duplicate values from the k1 column.

```
data.drop_duplicates(subset='k1')
```

	k1	k2
0	one	3
3	two	3

Now, we will learn to categorize rows based on a predefined criteria. It happens a lot while data processing where you need to categorize a variable. For example, say we have got a column with country names and we want to create a new variable 'continent' based on these country names. In such situations, we will require the steps below:

```
data    =    pd.DataFrame({'food':    ['bacon',    'pulled    pork',    'bacon',
'Pastrami','corned beef', 'Bacon', 'pastrami', 'honey ham','nova lox'],
                   'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
data
```

	food	ounces
0	bacon	4.0
1	pulled pork	3.0
2	bacon	12.0
3	Pastrami	6.0
4	corned beef	7.5
5	Bacon	8.0
6	pastrami	3.0
7	honey ham	5.0
8	nova lox	6.0

Now, we want to create a new variable which indicates the type of animal which acts as the source of the food. To do that, first we'll create a dictionary to map the food to the animals. Then, we'll use map function to map the dictionary's values to the keys. Let's see how is it done.

```
meat_to_animal = {
'bacon': 'pig',
'pulled pork': 'pig',
'pastrami': 'cow',
'corned beef': 'cow',
'honey ham': 'pig',
'nova lox': 'salmon'
}

def meat_2_animal(series):
    if series['food'] == 'bacon':
        return 'pig'
    elif series['food'] == 'pulled pork':
        return 'pig'
    elif series['food'] == 'pastrami':
        return 'cow'
    elif series['food'] == 'corned beef':
        return 'cow'
    elif series['food'] == 'honey ham':
        return 'pig'
    else:
        return 'salmon'


#create a new variable
data['animal'] = data['food'].map(str.lower).map(meat_to_animal)
data
```

	food	ounces	animal
0	bacon	4.0	pig
1	pulled pork	3.0	pig
2	bacon	12.0	pig
3	Pastrami	6.0	cow
4	corned beef	7.5	cow
5	Bacon	8.0	pig

	food	ounces	animal
6	pastrami	3.0	cow
7	honey ham	5.0	pig
8	nova lox	6.0	salmon

```
#another way of doing it is: convert the food values to the lower case and apply
the function
lower = lambda x: x.lower()
data['food'] = data['food'].apply(lower)
data['animal2'] = data.apply(meat_2_animal, axis='columns')
data
```

	food	ounces	animal	animal2
0	bacon	4.0	pig	pig
1	pulled pork	3.0	pig	pig
2	bacon	12.0	pig	pig
3	pastrami	6.0	cow	cow
4	corned beef	7.5	cow	cow
5	bacon	8.0	pig	pig
6	pastrami	3.0	cow	cow

	food	ounces	animal	animal2
7	honey ham	5.0	pig	pig
8	nova lox	6.0	salmon	salmon

Another way to create a new variable is by using the assign function. With this tutorial, as you keep discovering the new functions, you'll realize how powerful pandas is.

```
data.assign(new_variable = data['ounces']*10)
```

	food	ounces	animal	animal2	new_variable
0	bacon	4.0	pig	pig	40.0
1	pulled pork	3.0	pig	pig	30.0
2	bacon	12.0	pig	pig	120.0
3	pastrami	6.0	cow	cow	60.0
4	corned beef	7.5	cow	cow	75.0
5	bacon	8.0	pig	pig	80.0
6	pastrami	3.0	cow	cow	30.0
7	honey ham	5.0	pig	pig	50.0

	food	ounces	animal	animal2	new_variable
8	nova lox	6.0	salmon	salmon	60.0

Let's remove the column animal2 from our data frame.

```
data.drop('animal2',axis='columns',inplace=True)
data
```

	food	ounces	animal
0	bacon	4.0	pig
1	pulled pork	3.0	pig
2	bacon	12.0	pig
3	Pastrami	6.0	cow
4	corned beef	7.5	cow
5	Bacon	8.0	pig
6	pastrami	3.0	cow
7	honey ham	5.0	pig
8	nova lox	6.0	salmon

We frequently find missing values in our data set. A quick method for imputing missing values is by filling the missing value with any random number. Not just missing values, you may find lots of outliers in your data set, which might require replacing. Let's see how can we replace values.

```
#Series function from pandas are used to create arrays
data = pd.Series([1., -999., 2., -999., -1000., 3.])
data
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64


#replace -999 with NaN values
data.replace(-999, np.nan,inplace=True)
data
0       1.0
1       NaN
2       2.0
3       NaN
4   -1000.0
5       3.0
dtype: float64


#We can also replace multiple values at once.
data = pd.Series([1., -999., 2., -999., -1000., 3.])
data.replace([-999,-1000],np.nan,inplace=True)
data
0    1.0
1    NaN
2    2.0
3    NaN
4    NaN
5    3.0
dtype: float64
```

Now, let's learn how to rename column names and axis (row names).

```
data  =  pd.DataFrame(np.arange(12).reshape((3,  4)),index=['Ohio',  'Colorado',
'New York'],columns=['one', 'two', 'three', 'four'])
data
```

	one	**two**	**three**	**four**
Ohio	0	1	2	3

	one	two	three	four
Colorado	4	5	6	7
New York	8	9	10	11

```
#Using rename function
data.rename(index                          =                            {'Ohio':'SanF'},
columns={'one':'one_p','two':'two_p'},inplace=True)
data
```

	one_p	two_p	three	four
SanF	0	1	2	3
Colorado	4	5	6	7
New York	8	9	10	11

```
#You can also use string functions
data.rename(index = str.upper, columns=str.title,inplace=True)
data
```

	One_p	Two_p	Three	Four
SANF	0	1	2	3
COLORADO	4	5	6	7
NEW YORK	8	9	10	11

Next, we'll learn to categorize (bin) continuous variables.

```
ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

We'll divide the ages into bins such as 18-25, 26-35,36-60 and 60 and above.

```
#Understand the output - '(' means the value is included in the bin, '[' means
the value is excluded
bins = [18, 25, 35, 60, 100]
cats = pd.cut(ages, bins)
cats
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100],
(35, 60], (35, 60], (25, 35]]
Length: 12
Categories (4, object): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]


#To include the right bin value, we can do:
pd.cut(ages,bins,right=False)
[[18, 25), [18, 25), [25, 35), [25, 35), [18, 25), ..., [25, 35), [60, 100),
[35, 60), [35, 60), [25, 35)]
Length: 12
Categories (4, object): [[18, 25) < [25, 35) < [35, 60) < [60, 100)]


#pandas library intrinsically assigns an encoding to categorical variables.
cats.labels
array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)


#Let's check how many observations fall under each bin
pd.value_counts(cats)
(18, 25]     5
(35, 60]     3
(25, 35]     3
(60, 100]    1
dtype: int64
```

Also, we can pass a unique name to each label.

```
bin_names = ['Youth', 'YoungAdult', 'MiddleAge', 'Senior']
new_cats = pd.cut(ages, bins,labels=bin_names)

pd.value_counts(new_cats)
```

Youth	5
MiddleAge	3
YoungAdult	3
Senior	1
dtype: int64	

```
#we can also calculate their cumulative sum
pd.value_counts(new_cats).cumsum()
```

Youth	5
MiddleAge	3
YoungAdult	3
Senior	1
dtype: int64	

Let's proceed and learn about grouping data and creating pivots in pandas. It's an immensely important data analysis method which you'd probably have to use on every data set you work with.

```
df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
                   'key2' : ['one', 'two', 'one', 'two', 'one'],
                   'data1' : np.random.randn(5),
                   'data2' : np.random.randn(5)})
df
```

data1	data2	key1	key2
0	0.973599	0.001761	a
1	0.207283	-0.990160	a
2	1.099642	1.872394	b
3	0.939897	-0.241074	b
4	0.606389	0.053345	a

```
#calculate the mean of data1 column by key1
grouped = df['data1'].groupby(df['key1'])
grouped.mean()
key1
a    0.595757
b    1.019769
Name: data1, dtype: float64
```

Now, let's see how to slice the data frame.

```
dates = pd.date_range('20130101',periods=6)
df = pd.DataFrame(np.random.randn(6,4),index=dates,columns=list('ABCD'))
df
```

	A	B	C	D
2013-01-01	1.030816	-1.276989	0.837720	-1.490111

	A	B	C	D
2013-01-02	-1.070215	-0.209129	0.604572	-1.743058
2013-01-03	1.524227	1.863575	1.291378	1.300696
2013-01-04	0.918203	-0.158800	-0.964063	-1.990779
2013-01-05	0.089731	0.114854	-0.585815	0.298772
2013-01-06	0.222260	0.435183	-0.045748	0.049898

```
#get first n rows from the data frame
df[:3]
```

	A	B	C	D
2013-01-01	1.030816	-1.276989	0.837720	-1.490111
2013-01-02	-1.070215	-0.209129	0.604572	-1.743058
2013-01-03	1.524227	1.863575	1.291378	1.300696

```
#slice based on date range
df['20130101':'20130104']
```

	A	B	C	D
2013-01-01	1.030816	-1.276989	0.837720	-1.490111

	A	B	C	D
2013-01-02	-1.070215	-0.209129	0.604572	-1.743058
2013-01-03	1.524227	1.863575	1.291378	1.300696
2013-01-04	0.918203	-0.158800	-0.964063	-1.990779

```
#slicing based on column names
df.loc[:,['A','B']]
```

	A	B
2013-01-01	1.030816	-1.276989
2013-01-02	-1.070215	-0.209129
2013-01-03	1.524227	1.863575
2013-01-04	0.918203	-0.158800
2013-01-05	0.089731	0.114854
2013-01-06	0.222260	0.435183

```
#slicing based on both row index labels and column names
df.loc['20130102':'20130103',['A','B']]
```

	A	B
2013-01-02	-1.070215	-0.209129
2013-01-03	1.524227	1.863575

```
#slicing based on index of columns
df.iloc[3] #returns 4th row (index is 3rd)
A    0.918203
B   -0.158800
C   -0.964063
D   -1.990779
Name: 2013-01-04 00:00:00, dtype: float64


#returns a specific range of rows
df.iloc[2:4, 0:2]
```

	A	B
2013-01-03	1.524227	1.863575
2013-01-04	0.918203	-0.158800

```
#returns specific rows and columns using lists containing columns or row indexes
df.iloc[[1,5],[0,2]]
```

	A	C
2013-01-02	-1.070215	0.604572
2013-01-06	0.222260	-0.045748

Similarly, we can do Boolean indexing based on column values as well. This helps in filtering a data set based on a pre-defined condition.

```
df[df.A > 1]
```

	A	B	C	D
2013-01-01	1.030816	-1.276989	0.837720	-1.490111
2013-01-03	1.524227	1.863575	1.291378	1.300696

```
#we can copy the data set
df2 = df.copy()
df2['E']=['one', 'one','two','three','four','three']
df2
```

	A	B	C	D	E
2013-01-01	1.030816	-1.276989	0.837720	-1.490111	one
2013-01-02	-1.070215	-0.209129	0.604572	-1.743058	one
2013-01-03	1.524227	1.863575	1.291378	1.300696	two
2013-01-04	0.918203	-0.158800	-0.964063	-1.990779	three
2013-01-05	0.089731	0.114854	-0.585815	0.298772	four
2013-01-06	0.222260	0.435183	-0.045748	0.049898	three

```
#select rows based on column values
df2[df2['E'].isin(['two','four'])]
```

	A	B	C	D	E
2013-01-03	1.524227	1.863575	1.291378	1.300696	two
2013-01-05	0.089731	0.114854	-0.585815	0.298772	four

```
#select all rows except those with two and four
df2[~df2['E'].isin(['two','four'])]
```

	A	B	C	D	E
2013-01-01	1.030816	-1.276989	0.837720	-1.490111	one
2013-01-02	-1.070215	-0.209129	0.604572	-1.743058	one
2013-01-04	0.918203	-0.158800	-0.964063	-1.990779	three
2013-01-06	0.222260	0.435183	-0.045748	0.049898	three

We can also use a query method to select columns based on a criterion. Let's see how!

```
#list all columns where A is greater than C
df.query('A > C')
```

	A	B	C	D
2013-01-01	1.030816	-1.276989	0.837720	-1.490111
2013-01-03	1.524227	1.863575	1.291378	1.300696

	A	B	C	D
2013-01-04	0.918203	-0.158800	-0.964063	-1.990779
2013-01-05	0.089731	0.114854	-0.585815	0.298772
2013-01-06	0.222260	0.435183	-0.045748	0.049898

```
#using OR condition
df.query('A < B | C > A')
```

	A	B	C	D
2013-01-02	-1.070215	-0.209129	0.604572	-1.743058
2013-01-03	1.524227	1.863575	1.291378	1.300696
2013-01-05	0.089731	0.114854	-0.585815	0.298772
2013-01-06	0.222260	0.435183	-0.045748	0.049898

Pivot tables are extremely useful in analyzing data using a customized tabular format. I think, among other things, Excel is popular because of the pivot table option. It offers a super-quick way to analyze data.

```
#create a data frame
data = pd.DataFrame({'group': ['a', 'a', 'a', 'b','b', 'b', 'c', 'c','c'],
                 'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
data
```

	group	ounces
0	a	4.0
1	a	3.0
2	a	12.0
3	b	6.0
4	b	7.5
5	b	8.0
6	c	3.0
7	c	5.0
8	c	6.0

```
#calculate means of each group
data.pivot_table(values='ounces',index='group',aggfunc=np.mean)
group
a    6.333333
b    7.166667
c    4.666667
Name: ounces, dtype: float64


#calculate count by each group
data.pivot_table(values='ounces',index='group',aggfunc='count')
group
a    3
b    3
c    3
```

```
Name: ounces, dtype: int64
```