To apply your own or another library’s functions to Pandas objects, you should be aware of the three important methods. The methods have been discussed below. The appropriate method to use depends on whether your function expects to operate on an entire DataFrame, row- or column-wise, or element wise.

* Table wise Function Application: pipe( )
* Row or Column Wise Function Application: apply( )
* Element wise Function Application: applymap( )

**Table-wise Function Application**

Custom operations can be performed by passing the function and the appropriate number of parameters as pipe arguments. Thus, operation is performed on the whole DataFrame.

# creating a dataframe (table):

import pandas as pd

import numpy as np

df = pd.DataFrame([[1,2], [3,4],[5,6]], index = ['x','y','z'], columns = ['a','b'])

print(df)

o/p:   
 a b

x 1 2

y 3 4

z 5 6

# adding a value 2 to all the elements in the dataframe

def adder(ele1,ele2):

return ele1+ele2

df = pd.DataFrame([[1,2], [3,4],[5,6]], index = ['x','y','z'], columns = ['a','b'])

df2 = df.pipe(adder,2)

print(df2)

o/p:

a b

x 3 4

y 5 6

z 7 8

# multiplying all the elements by 10 in the dataframe

def product(ele1,ele2):

return ele1\*ele2

df = pd.DataFrame([[1,2], [3,4],[5,6]], index = ['x','y','z'], columns = ['a','b'])

print(df)

df2 = df.pipe(product,10)

print(df2)

o/p:

a b

x 10 20

y 30 40

z 50 60

## Row or Column Wise Function Application

Arbitrary functions can be applied along the axes of a DataFrame or Panel using the **apply()** method, which, like the descriptive statistics methods, takes an optional axis argument. By default, the operation performs column wise, taking each column as an array-like.

# displaying the mean value column wise using apply()

df = pd.DataFrame([[1,2], [3,4],[5,6]], index = ['x','y','z'], columns = ['a','b'])

print(df)

df2 = df.apply(np.mean)

print(df2)

o/p:

a 3.0

b 4.0

dtype: float64

By default, the apply() performs column wise operation, taking each column as an array-like.

# displaying the mean value row wise using apply()

df = pd.DataFrame([[1,2], [3,4],[5,6]], index = ['x','y','z'], columns = ['a','b'])

print(df)

df2 = df.apply(np.mean, axis=1)

print(df2)

o/p:

x 1.5

y 3.5

z 5.5

dtype: float64

# max and min functions

df = pd.DataFrame([[1,2], [3,4],[5,6]], index = ['x','y','z'], columns = ['a','b'])

print(df)

df2 = df.apply(lambda x: x.max() - x.min()) #column wise

print(df2)

o/p:

a 4

b 4

dtype: int64

df2 = df.apply(lambda x: x.max() - x.min(), axis=1) #row wise

print(df2)

o/p:

x 1

y 1

z 1

dtype: int64

## Element Wise Function Application

Not all functions can be vectorized (neither the NumPy arrays which return another array nor any value), the methods **applymap()** on DataFrame and **analogously map()** on Series accept any Python function taking a single value and returning a single value.

# multiplying each column of a dataframe [series] with a certain value using map()

When you are performing any operation column wise, here the column is considered as a Series. We use the Map() to perform any operation on entire column wise.

df = pd.DataFrame([[1,2], [3,4],[5,6]], index = ['x','y','z'], columns = ['a','b'])

print(df)

df2 = df['a'].map(lambda p: p\*10)

df3 = df['b'].map(lambda p: p\*10)

print(df2)

print(df3)

o/p:

x 10

y 30

z 50

Name: a, dtype: int64

x 20

y 40

z 60

Name: b, dtype: int64

# performing operations on entire dataframe by a certain value using applymap()

When you are performing any operation on an entire dataframe, we use the applymap()

df = pd.DataFrame([[1,2], [3,4],[5,6]], index = ['x','y','z'], columns = ['a','b'])

print(df)

df2 = df.applymap(lambda p: p\*10)

print(df2)

o/p:

a b

x 10 20

y 30 40

z 50 60

# multiplying each row of a dataframe by a different number

df = pd.DataFrame([[1,2], [3,4]], index = ['x','y'], columns = ['a','b'])

print(df)

df2 = df.mul([10, 100], axis=0)

print(df2)

o/p:

a b

x 10 20

y 300 400

df = pd.DataFrame([[1,2], [3,4]], index = ['x','y'], columns = ['a','b'])

print(df)

df2 = df.add([10,20], axis=1)

df3 = df.add([10,20], axis=0)

print(df2)

print(df3)

o/p:

a b

x 11 22

y 13 24

a b

x 11 12

y 23 24

# multiply two dataframes

df1 = pd.DataFrame([[2,2], [2,2]], index = ['x','y'], columns = ['a','b'])

print(df1)

df2 = pd.DataFrame([[4,4], [4,4]], index = ['p','q'], columns = ['c','d'])

print(df2)

df3 = pd.np.multiply(df1,df2)

print(df3)

o/p:

a b

x 2 2

y 2 2

c d

p 4 4

q 4 4

a b

x 8 8

y 8 8

# creating a dataframe

df = pd.DataFrame(np.array([[1,2,3],[4,5,6],[7,8,9]]), columns=[‘A’,’B’,’C’])

print(df)

o/p:

A B C

0 1 2 3

1 4 5 6

2 7 8 9

print(df.shape) ---------------🡪 (3,3)

print(df.size) ---------------🡪 9

print(len(df.index)) ---------🡪 3

You could also use df[0].count() to get to know more about the height of your DataFrame, but this will exclude the NaN values (if there are any). That is why calling .count() on your DataFrame is not always the better option.

# to display the columns:

print(list(df.columns.values))

o/p: [‘A’, ‘B’,’C’]

# to display rows (index):

print(list(df.index.values))

o/p: [0,1,2]

# indexing

print(df.iloc[0][0]) -------------🡪 1

print(df.loc[0][0]) -------------🡪 1

print(df.loc[1]['B']) ----------🡪 5

print(df.loc[2]['C']) -----------🡪 9

print(df.at[0,'A']) ------------🡪 1

print(df.iat[0,0]) -------------🡪 1

# selecting values in a particular row

print(df.iloc[2])

o/p:

A 7

B 8

C 9

Name: 2, dtype: int64

#selecting values in a particular column

print(df.loc[1:, 'C'])

o/p:

1 6

2 9

Name: C, dtype: int64

Difference between .iloc(), .loc() and .ix()

* .loc[] works on labels of your index. This means that if you give in loc[2], you look for the values of your DataFrame that have an index labeled 2.
* .iloc[] works on the positions in your index. This means that if you give in iloc[2], you look for the values of your DataFrame that are at index ’2`.
* .ix[] is a more complex case: when the index is integer-based, you pass a label to .ix[]. ix[2] then means that you’re looking in your DataFrame for values that have an index labeled 2. This is just like .loc[]! However, if your index is not solely integer-based, ix will work with positions, just like .iloc[].

ix[] works on indexes. .ix[] will have the same behavior as iloc and look at the positions in the index.

print(df.loc[2])

print(df.iloc[2])

print(df.ix[2])

print(df.ix['A'])

print(df.ix[1])

o/p:

48 1

49 2

50 3

Name: 2, dtype: int64

48 7

49 8

50 9

Name: 4, dtype: int64

48 7

49 8

50 9

Name: 4, dtype: int64

48 4

49 5

50 6

Name: A, dtype: int64

48 4

49 5

50 6

Name: A, dtype: int64

print(df.loc[2][49]) -------------🡪 2

print(df.iloc[2][48]) -----------🡪 7

print(df.ix[2][50]) --------------🡪 9

Tip:  the general recommendation is that you use .loc to insert rows in your DataFrame. That is because if you would use df.ix[], you might try to reference a numerically valued index with the index value and accidentally overwrite an existing row of your DataFrame.

df = pd.DataFrame(data=np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), index= [2.5, 12.6, 4.8], columns=[48, 49, 50])

print(df)

o/p:

48 49 50

2.5 1 2 3

12.6 4 5 6

4.8 7 8 9

df.ix[2] = [60, 50, 40]

print(df)

o/p:

48 49 50

2.5 1 2 3

12.6 4 5 6

4.8 60 50 40

df.loc[2] = [11, 12, 13]

print(df)

o/p:

48 49 50

2.5 1 2 3

12.6 4 5 6

4.8 7 8 9

2.0 11 12 13

### Adding an Index to a DataFrame

When you create a DataFrame, you have the option to add input to the ‘index’ argument to make sure that you have the index that you desire. When you don’t specify this, your DataFrame will have, by default, a numerically valued index that starts with 0 and continues until the last row of your DataFrame.

However, even when your index is specified for you automatically, you still have the power to re-use one of your columns and make it your index. You can easily do this by calling set\_index() on your DataFrame.

print(df.set\_index('C'))  
o/p:

A B

C

3 1 2

6 4 5

9 7 8

### Adding a Column to Your DataFrame

In some cases, you want to make your index part of your DataFrame. You can easily do this by taking a column from your DataFrame or by referring to a column that you haven’t made yet and assigning it to the .index property.

df = pd.DataFrame(data=np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), columns=['A', 'B', 'C'])

print(df)

df['D'] = df.index

print(df)

o/p:

A B C

0 1 2 3

1 4 5 6

2 7 8 9

A B C D

0 1 2 3 0

1 4 5 6 1

2 7 8 9 2

# alternate way to add a column

df.loc[:, 'D'] = pd.Series([10,11,12], index = df.index)

print(df)

### Resetting the Index of Your DataFrame

When your index doesn’t look entirely the way you want it to, you can opt to reset it. You can easily do this with .reset\_index(). However, you should still watch out, as you can pass several arguments that can make or break the success of your reset:

# Check out the weird index of your dataframe

print(df)

o/p:

48 49 50

2.5 1 2 3

12.6 4 5 6

4.8 7 8 9

# Use `reset\_index()` to reset the values.

df\_reset = df.reset\_index(level=0, drop=True)

print(df\_reset)

o/p:

48 49 50

0 1 2 3

1 4 5 6

2 7 8 9

Now try replacing the drop argument by inplace in the code chunk above and see what happens!

Note how you use the drop argument to indicate that you want to get rid of the index that was there. If you would have used inplace, the original index with floats is added as an extra column to your DataFrame.