Python for Data Analysis

By……. Sakeeb Sheikh

Lambda Function

lambda operator or lambda function is used for creating small, one-time and anonymous function objects in Python.

Basic syntax:

```
lambda arguments : expression
```

lambda operator can have any number of arguments, but it can have only one expression.

It cannot contain any statements and it returns a function object which can be assigned to any variable.

Example 1 :

Function in python:

```
def add(x, y):
        return x + y

# Call the function
add(2, 3)
# Output: 5
```

Example 1 :

```
add = lambda x, y : x + y
print(add(2, 3))
```

Output: 5

Note:- Mostly lambda functions are passed as parameters to a function which expects a function objects as parameter like map, reduce, filter functions

Map function :

Basic syntax :

map(function_object, iterable1, iterable2,...)

- *map* functions expects a function object and any number of **iterables** like list, dictionary, etc.
- It executes the function_object for each element in the sequence and returns a **list of the elements modified by the function object**

Example :-

```
def multiply2(x):
    return x * 2

map(multiply2, [1, 2, 3, 4])
# Output [2, 4, 6, 8]
```

Writing with function using lamda function

```
map(lambda x : x*2, [1, 2, 3, 4]) #Output [2, 4, 6, 8]
```

Map with multiple Iterables

```
list_a = [1, 2, 3]
list_b = [10, 20, 30]

map(lambda x, y: x + y, list_a, list_b)
# Output: [11, 22, 33]
```

Need to Convert to list cannot access element directly

```
list_map_output = list(map_output)
```

Filter() function

Basic syntax :-

filter(function_object, iterable)

- *filter* function expects **two arguments**, function_object and an iterable.
- function_object **returns a boolean value**. function_object is called for each element of the iterable and filter returns only those **element** for which the **function_object returns *true***.

Example : filter()

```
a = [1, 2, 3, 4, 5, 6]
 filter_obj = filter(lambda x : x % 2 == 0, a)
# Output: [2, 4, 6]
```

- Cannot access elements nor find the len() of the list
- Need to convert to list

```
# Converts the filer obj to a list
even_num = list(filter_obj)
```

Vectorizing functions (vectorize)

suppose you have a Python function named addsubtract defined as

```
def addsubtract(a,b):
...    if a > b:
...        return a - b
...    else:
...        return a + b
```

the class vectorize can be used to “vectorize “ this function so that

```
vec_addsubtract = np.vectorize(addsubtract)
```

returns a function which takes array arguments and returns an array result

```
vec_addsubtract([0,3,6,9],[1,3,5,7])
```

Select() Function

select which extends the functionality of **where()** to include multiple conditions and multiple choices. select is a vectorized form of the multiple if-statement.

```
x = np.arange(-2,3)
print(x)
array([-2, -1,  0,  1,  2])
y = np.select([x > 3, x >= 0], [0, x+2])
print(y)
```

Concatenation of multiple Dataframes

```
import pandas as pd


df1 = pd.read_csv('../data/concat_1.csv')
df2 = pd.read_csv('../data/concat_2.csv')
df3 = pd.read_csv('../data/concat_3.csv')
```

concatenate rows

```
row_concat = pd.concat([df1, df2, df3])

Print(row_concat)

row_concat.loc[0]

row_concat.iloc[0]
```

```
# create a new row to concatenate
new_row = pd.Series(['n1', 'n2', 'n3', 'n4'])
new_row

# Examine the output what it gives
pd.concat([df1, new_row])
```

Correct way to Concatenate

```
# note the double brackets
new_row_2 = pd.DataFrame([['n1', 'n2', 'n3', 'n4']],
                         columns = ['A', 'B', 'D', 'C'])
Print(new_row_2)
#If we pass less number of columns
new_row_3 = pd.DataFrame([['n1', 'n2', 'n3']],
                         columns = ['A', 'B', 'D'])
Print(new_row_3)
pd.concat([df1, new_row_3])  #Fills the unspecified columns with NaN
```

Concatenate Columns

```
col_concat = pd.concat([df1, df3, df3], axis=1)
print(col_concat)
```

Concatenation with different indicies

df1.columns = ['A', 'B', 'C', 'D']

df2.columns = ['E', 'F', 'G', 'H']

df3.columns = ['A', 'C', 'F', 'H']

pd.concat([df1, df2, df3])

Changing the row index

- df1.index = range(4) # 0 to 3 inclusive
- df2.index = range(4, 8) # 4 to 7 inclusive
- df3.index = [0, 2, 5, 7]

merge() (kind of join operation in sql)

- Loading Some Datasets

```
person = pd.read_csv(‘survey_person.csv')
site = pd.read_csv('survey_site.csv')
survey = pd.read_csv('survey_survey.csv')
visited = pd.read_csv(‘survey_visited.csv')

print(person)
print(site)
print(survey)
print(visited)
```

Slicing the Dataframe using .iloc[row:column]

```
visited_subset = visited.iloc[[0, 2, 6], :]
Print(visited_subset)

# a one-to-one merge
o2o = pd.merge(left=site, right=visited_subset,
               left_on='name', right_on='site')
Print(o2o)
```

Another way to merge

```
# a many-to-one merge
# note the different way to perform the merge
m2o = site.merge(visited, left_on='name', right_on='site')
m2o
```

Another way to merge

```
# many to many
# cartesian product
df1 = pd.DataFrame({
    'a': [1, 1, 1, 2, 2],
    'b': [10, 20, 30, 40, 50]
})

df2 = pd.DataFrame({
    'a1': [1, 1, 2, 2, 3],
    'b2': [100, 200, 300, 400, 500]
})
df1.merge(df2, left_on='a', right_on='a1')
```

Handling missing / NaN Values?

```
visited = pd.read_csv('survey_visited.csv')
survey = pd.read_csv('survey_survey.csv')

vs = visited.merge(survey, left_on='ident', right_on='taken')
Print(vs)
```

count missing data

```
ebola = pd.read_csv('ebola_country_timeseries.csv')

ebola.info()

# sorts by frequency
# will not always be on top
ebola['Cases_Guinea'].value_counts(dropna=False).head()
```

Recode missing

```
ebola.head()

ebola.fillna(0).head()

ebola.fillna(method='ffill').head()

ebola.fillna(method='ffill').tail()

ebola.fillna(method='bfill').head()

ebola.fillna(method='bfill').tail()
```

Calculations with missing

→ **Without NaN Values**

```
ebola['Cases_Guinea'].sum()
```

→ **With NaN Values**

```
ebola['Cases_Guinea'].sum(skipna=False)
```

Working With Groupby Function

```
gapminder = pd.read_csv('../datasets/gapminder.tsv', sep='\t')

gapminder.head()

#finding the mean() year wise
gapminder.groupby('year')['lifeExp'].mean()
```

 # breaking the groupby down

```
y1952 = gapminder.loc[gapminder['year'] == 1952, :]
y1952.head()

y1952['lifeExp'].mean()
```

methods you can use

- count
- size
- mean
- std
- min
- quantile(q=0.25)
- max
- sum
- var
- sem
- describe
- first
- last
- nth

Using describe() function

```
gapminder.groupby('continent')['lifeExp'].describe()
```

Using agg() and aggregate()

```
## use agg to call functions from other libraries
## or even functions you write yourself
import numpy as np

# these 2 do the same thing
gapminder.groupby('continent')['lifeExp'].aggregate(np.mean)
gapminder.groupby('continent')['lifeExp'].agg(np.mean)
```

Using UserDefined Function

```
def my_mean(values):
    n = len(values)
    s = np.sum(values)
    return s / n


gapminder.groupby('continent')['lifeExp'].agg(my_mean)
```

Using Multiple UserDefined Functions

```
# multiple functions
gapminder.groupby('year')['lifeExp'].agg([
    np.count_nonzero,
    np.mean,
    np.std
])
```

Understanding apply() function

Writing Some Basic Functions :-

First funtion :-

```
def my_function():
        pass  # used for a completely empty function
```

Second Function :-

```
def my_sq(x):
    """Squares a given value
    """
    return x ** 2
my_sq(4)
```

```
def avg_2(x, y):
    """Calculates the average between 2 numbers
    """
    return (x + y) / 2  # note that this will be 2.0 in Python 2

avg_2(10, 20)
```

apply basic

```
df = pd.DataFrame({
    'a': [10, 20, 30],
    'b': [20, 30, 40]
})
Df


# let's use our square function
# yes we could've also done the calculation directly on the column
df['a'] ** 2
```

```
df['a'].apply(my_sq)

def my_exp(x, e):
    '''raise x to the e power
    '''
    return x ** e

# apply a function by providing multiple arguments
df['a'].apply(my_exp, e=2)
df['a'].apply(my_exp, e=3)
```

Apply over a dataframe

```
def print_me(x):
    """just prints what you give it
    """
    print(x)
```

column-wise

```
df.apply(print_me)
```

```
def avg_3(x, y, z):
    """Calculate average between 3 numbers
    """
    return (x + y + z) / 3

# use the function on each column
# this will cause an error
df.apply(avg_3)
```

Another way without errors

```
# have to re-write the function
# in an apply, the entire column (or row)
# get's passed in to the first agument
def avg_3_apply(col):
    """Calculate average between 3 numbers
    """
    x = col[0]
    y = col[1]
    z = col[2]
    return (x + y + z) / 3
df.apply(avg_3_apply)
```

row-wise

```
# we can't just use the same function as before
df.apply(avg_3_apply, axis=1)  #This will give error


def avg_2_apply(row):
    x = row[0]
    y = row[1]
    return (x + y) / 2
df.apply(avg_2_apply, axis=1)
```

Python Pandan’s DataFrame Summary