My Master Code Book & FAQ's

Table of contents

Basics

Python Basics

```
# To extract all keywords in python 3.6 use the below code

import keyword

# Import basic packages
import numpy as np                                          # Implemennts milti-
dimensional array and matrices
import pandas as pd                                         # For data manipulation and
analysis
import pandas_profiling
import matplotlib.pyplot as plt                             # Plotting library for Python
programming language and it's numerical mathematics extension NumPy
import seaborn as sns                                       # Provides a high level
interface for drawing attractive and informative statistical graphics
%matplotlib inline
sns.set()

from subprocess import check_output

from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))
```

```
More samples
```

```
from IPython.display import Image
Image(filename='fig/img_4926.jpg')

from IPython.display import YouTubeVideo
YouTubeVideo('iwVvqwLDsJo')
```

```
#borders for dataframe

%%HTML
<style type="text/css">
    table.dataframe td, table.dataframe th {
        border-style: solid;
        border: 3px solid lightgray;
    }
</style>
```

```
#TOGGLE YOUR RAW CODES

from IPython.display import HTML

HTML('''<script>
code_show=true;
function code_toggle() {
 if (code_show){
 $('div.input').hide();
 } else {
 $('div.input').show();
 }
 code_show = !code_show
}
$( document ).ready(code_toggle);
</script>
<form action="javascript:code_toggle()"><input type="submit" value="Click here to toggle
on/off the raw code."></form>''')
```

```
%%HTML
#Writing Docstring

def square(num):
    """
    Function to square a number.
    Print this docstring using __doc__ attribute of the function
    """
    return num * num

print(square(20))
print(square.__doc__)
```

```
print(keyword.kwlist)        #all keyword list
print(id(Add your variable here))               #print address of variable:
```

```
Integers, floating point numbers and complex numbers falls under Python numbers category.
They are defined as int, float and complex class in Python.

We can use the type() function to know which class a variable or a value belongs to
and the isinstance() function to check if an object belongs to a particular class.
```

```
Python Strings
String is sequence of Unicode characters. We can use single quotes or double quotes to
represent strings.
Multi-line strings can be denoted using triple quotes(single/double), ''' or """.
A string in Python consists of a series or sequence of characters - letters, numbers, and
special characters.
Strings can be indexed - often synonymously called subscripted as well.
The first character of a string has the index 0.
Slicing string
hey = "Om is great"
print("What do you know?:",hey)
print("who is great?:", hey[0:2])
print("Om is what? ", hey[6:12])
```

```
Python List
List is an ordered sequence of items. It is one of the most used datatype in Python and
is very flexible.
All the items in a list do not need to be of the same type.
Declaring a list is pretty straight forward. Items separated by commas are enclosed
within square brackets '[ ]'.

print(myList[2])                     # Print an element based on its index. Index starts
from 0
myList.remove(2.2)                            # Remove item from a particular index
print(myList)
LIST.pop(3)                                # Remove item from a particular index
myList.append(2.2)                           # Add item at the last index
```

```
Python Tuple

Tuple is an ordered sequence of items same as list.
The only difference is that tuples are immutable. Tuples once created cannot be modified.
Tuples are used to write-protect data and are usually faster than list as it cannot
change dynamically.
It is defined within parentheses () where items are separated by commas.

myTuple = (10,20,30,"Text")
print(myTuple[0:3])                 # Read elements by their corresponding index values
```

```
Python Set
Set is an unordered collection of unique items.
Set is defined by values separated by comma inside curly braces { }.

mySet = {10, 20, 30, 40, 50, 0}
mySet = {10, 20, 20, 30, 30,40}       # Only unique values considered, duplicates removed
automatically
print(mySet)
OUT - {40, 10, 20, 30}
```

```
Python Dictionary
Dictionary is an unordered collection of key-value pairs.

It is generally used when we have a huge amount of data.
Dictionaries are optimized for retrieving data.
We must know the key to retrieve the value.

In Python, dictionaries are defined within curly braces {}
with each item being a pair in the form key:value.
```

```
Key and value can be of any type.
test = {'key1': "value1", 'key2': "value2", 'key3': "value3",}

Update a value in the dictionary
myDictionary['key2'] = "value2.2"
```

```
Conversion between Datatypes
To convert between different data types use different type conversion functions like
int(), float(), str() etc.

# 1> list to set
myList1 = ['a','b','c','c','d','D']
print(myList1)
print(type(myList1))
mySet1 = set(myList1)
print(type(mySet1))
print(mySet1)

# 2> String to list
myString1 = 'This is string to list'
print(type(myString1))

om = list(myString1)
print(om, type(om))
print(om[0:4])
om1 = str(om)
print(om1)
print(om[0])
```

```
List Comprehensions
List comprehensions provide a concise way to create lists.
This can be thought of a process that makes it easier to create lists.

Common applications are to make new lists where each element is the result of some
operations applied to each member of another sequence or iterable, or to create a
subsequence of those elements that satisfy a certain condition.

myList = []               # Normal way to create an empty List

for i in range(4,20): # Append the squared element to the list
    myList.append(i**3)
print(myList)

myList = [i**2 for i in range(4)]
print(myList)

samelist = []
for i in range(5):
    rowlist = []
    for j in range(1,6):
        rowlist.append(j)
    samelist.append(rowlist)
print(samelist)
```

```
Output Formatting
print("{} is half of {}".format(myInt1, myInt2), "is incorrect statement")

Input
userInput = input("Please enter some data: ")
print("You typed in: ",userInput)
```

```
variable1, variable2 = 5, 2
print(variable1 + variable2)  # Addition(+)
print(variable1 - variable2)  # Subtraction(-)
print(variable1 * variable2)  # Multiplication(*)
print(variable1 / variable2)  # Division(/)
print(variable1 % variable2)  # Modulo division (%)
print(variable1 // variable2) # Floor Division (//)
print(variable1 ** (variable2*3)) # Exponent (**)
```

```
variable1, variable2 = 3, 7            # Decimal to bitwise values -> 3 - 0000 0011 & 7 -
0000 0111
print(variable1 & variable2)           # Bitwise AND ->       0000 0011 & 0000 0111 ->
0000 0011 -> 3
print(variable1 | variable2)           # Bitwise OR ->        0000 0011 | 0000 0111 ->
0000 0111 -> 7
print(~variable2)                      # Bitwise NOT ->     ~ 0000 0111 -> 1111 1000 ->
-8
print(variable1 ^ variable2)           # Bitwise XOR ->       0000 0011 ^ 0000 0111 ->
0000 0100 -> 4
print(variable1>>2)                    # Bitwise rightshift   0000 0011>>2 -> 0000 0000 -
> 0
print(variable1<<2)                    # Bitwise Leftshift    0000 0011<<2 -> 0000 1100 -
> 12
```

```
Assignment operators
Assignment operators are used in Python to assign values to variables.

age = 50 is a simple assignment operator that assigns the value 50 on the right to the
variable (age) a on the left.
=,  +=,  -=,  *=,  /=,  %=,  //=,  **=, &=,  |=,  ^=,  >>=,  <<= are Assignment operators
age = 40
om = 55
age += 4        # Add AND  <- age = age + 4
age -= 7        # Subtract AND (-=)
age *= 4        # Multiply AND (*=)
age /= 4        # Divide AND (/=)
age %= 5       # Modulus AND (%=)
print(age)
om //= 11 # Floor Division (//=)
print(om)
om **= 2       # Exponent AND (**=)
print(om)
```

```
If else loop:

age = input()
age = int(age)
if age < 18:
    if age >= 12:
        print("Teen")
    else:
        print("child")
else:
    print("adult")
print('This will always get executed')
```

```
while loop: Use while loop to iterate over a block of code as long as the test expression
(also called test condition) is true.
```

```
Syntax
while test_expression:

    Body of while
The body of the loop is entered only if the test_expression evaluates to True.
After one iteration, the test expression is checked again.
This process continues until the test_expression evaluates to False.
While loop has an optional else block which one may use if one wishes to use it.
The else block gets executed when the condition in while statement is False.

The else can be skipped if we use a break command in the while block
myList = [1, 2, 3, 4, 5]

#iterating over the list
index = 0
while index < len(myList):
    print(myList[index])
    break
    index += 1
else:
    print('Completed iterating the list')

print('Eitherway printed')
```

```
for Loop Python: for loop in one of the most often used Python looping technque to
iterate over a sequence (list, tuple, string) or other iterable objects.
Iterating over a sequence is called traversal.
Syntax:
for element in sequence :
    'for' code block
Here, element is the variable that takes the value of the item inside the sequence on
each iteration.
Loop continues until we reach the last item in the sequence.

# Sum of all numbers in a list

myList = [1, 2, 3, 4, 5,2.5]

sum = 0
for values in myList:
    sum += values
print(sum)


range(start,stop,step size)
# Print range of 5
for element in range(5,10,2):
    print(element)
```

```
Examples of break and continue

# Use of break
names = ['Suchit', 'Rakesh', 'Roshni']
for name in names:               # Let us iterate over the names list
    if name == 'Rakesh':
        break
    print(name)
else:
    print('For is completed and we are in else part')
print("Totally outside the for & else loop")

# Use of continue
names = ['Suchit', 'Rakesh', 'Roshni']

for name in names:                # Let us iterate over the names list
    if name == 'Rakesh':
        continue
    print(name)                   # If continue condition is satisfied we skip this line
and carry on with the next iteration
else:
    print('For is completed and we are in else part')
print("Totally outside the for & else loop")
```

```
Python Set Operations

Operations:
union
intersection
symmetric difference
subset

mySet1 = {1, 2, 3, 4, 5}
mySet2 = {3, 4, 5, 6, 7}
print(mySet1 | mySet2)              # union of 2 sets using | operator
print(mySet1.union(mySet2))         # Alternately use union() method

print(mySet1 & mySet2)              # Intersection of 2 sets using & operator
print(mySet1.intersection(mySet2))  # Alternately use intersection() method

print(mySet1 - mySet2)              # set Difference: set of elements that are only in
mySet1 but not in mySet2
print(mySet1.difference(mySet2))    # use differnce() function method

print(mySet1^mySet2)                       # For symmetric difference use ^ operator
print(mySet1.symmetric_difference(mySet2)) # Alternately use symmetric_difference
function
```

```
Function

"def" keyword notifies the start of function header
Arguments (parameters) through which we pass values to a function. These are optional
A colon(:) to mark the end of funciton header
Doc string describe what the function does. This is optional
"return" statement to return a value from the function. This is optional
def Facto(num):
    """
    Fact of number
    """
    fact = 1
    while(num>0):
        fact *= num
        num -= 1
    return fact
number = 6

print("Factorial of number {} is : {}".format(number, Facto(number)) )
```

```
Builtin functions
```

```
all()¶
The function all() retruns,
True: If all elements in an iterable data collection are true
False: If any element in an iterable data collection is false (Remember 0 & None are
considered False)

dir()¶
The dir() tries to return a list of valid attributes of the object.
If the object has dir() method, the method will be called and must return the list of
attributes.
If the object doesn't have dir() method, this method tries to find information from the
dict attribute (if defined), and from type object. In this case, the list returned from
dir() may not be complete.

divmod()
The divmod() method takes two numbers and returns a pair of numbers (a tuple) consisting
of their quotient and remainder.

enumerate()
enumerate() method adds counter to an iterable data collection & returns it
Syntax: enumerate(iterable, start=0)
The enumerate() method takes two parameters:
iterable - a sequence, an iterator, or objects that supports iteration
start (optional) - enumerate() starts counting from this number. If start is omitted, 0
is taken as start.

filter()
The filter() method constructs an iterator from elements of an iterable for which a
function returns true.
Syntax: filter(function, iterable)
The filter() method takes two parameters:
function - function that tests if elements of an iterable returns true or false If None,
the function defaults to Identity function - which returns false if any elements are
false
iterable - iterable which is to be filtered, could be sets, lists, tuples, or containers
of any iterators

isinstance()
The isinstance() function checks if the object (first argument) is an instance or
subclass of classinfo class (second argument).
Syntax: isinstance(object, classinfo)

map()
Map applies a function to all the items in an input_list.
Syntax: map(function_to_apply, list_of_inputs)
```

```
Lambda or Anonymous Functions
In Python, anonymous function is a function that is defined without a name.
While normal functions are defined using def keyword,
in Python anonymous functions are defined using the lambda keyword.
Lambda functions are used extensively along with built-in functions like filter(), map()

square = lambda x: x**2          # Using lambe we write the squaring function as beside
print(square(10))

myList = [1, 2, 3, 4, 5]
oddList = list(filter(lambda x: (x%2 != 0), myList))
print(oddList)
```

Modules

Python provides a lot of standard modules that can be used for various purposes.
https://docs.python.org/3/py-modindex.html

```
import math
math.factorial(5)

import random
random.random()

from datetime import date
print(date.today())
```

File Handling ##Pending

Numpy

Introduction to Numpy
Numpy is a library developed for Python which can handle large, multi-dimensional arrays and matrices. It has a large collections of mathematical functions to operate on these arrays.

Lets have a brief comparison of Numpy with Python Lists.
Numpy is remarkably faster than Python Lists for many reasons.

It was designed for efficient data storage. All the elements of numpy arrays are stored sequentially with a fixed width for each value. On the other hand Lists are pointers to data stored elsewhere. The number of separate reads the computer has to do is smaller for numpy.

Numpy has uniform datatypes instead of Lists. The computer performs a logic for each different element type. This is completely avoided with Numpy.

```
Numpy has optimized functions for many mathematical operations on arrays and matrices.
This is why they are faster than regular math operations on lists.

import numpy as np

DATA = [[1,2,3],[4,5,6],[7,8,9],[10,11,12]]
type(DATA)
mat.shape
mat.dtype

arange(start, stop, step)
np.arange(0,11,5)
```

```
numpy.linspace() returns evenly spaced numbers over a specified interval
numpy.eye is used to generate an identity matrix
numpy.zeros generates a matrix with all elements 0.

np.zeros((4,2))
np.ones((12,218))

Generate 10 points between 0 and 5.
np.linspace(0,5,10)      #linspace is used for making high resolution plot

Generate an array of 5 random numbers
np.random.rand(5)

Generate a 4*4 matrix where the elements are distributed in random distribition.
np.random.randn(4,4)

np.reshape(#,#)

Array Slicing
To access more than one element of the array use slicing.

IN: test = np.array([[0,1,2],[3,4,5],[6,7,8],[9,10,11]])
OUT: array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])

IN: test[0:2,0:2]
OUT: array([[0, 1],
       [3, 4]])

IN: arr_2d = np.array([[1,2,3],[5,6,7],[8,9,10],[12,13,14]])    # Creating numpy array
arr_2d

OUT: array([[ 1,  2,  3],
       [ 5,  6,  7],
       [ 8,  9, 10],
       [12, 13, 14]])

IN: scaler = 3
IN: arr_2d +  3                                          # operation with a scaler
OUT: array([[ 4,  5,  6],
       [ 8,  9, 10],
       [11, 12, 13],
       [15, 16, 17]])

IN: arr_1d = np.array([10,10,10])
```

```
IN: arr_2d + arr_1d                                              # operation with a array
of different shape
OUT: array([[11, 12, 13],
       [15, 16, 17],
       [18, 19, 20],
       [22, 23, 24]])
```

Numpy Mathematical Functions

Here you can see a lot of commonly used built-in functions of numpy for mathematical operations. These functions are faster and optimized for large size arrays.

```
IN: arr = np.arange(1,11)  # Lets first create a numpy array
arr

OUT: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])

IN: arr.min()   = 2
IN: arr.max()  = 10

IN: arr.argmin()   = 0                                       # Index position of minimum
of array
IN: arr.argmax()   = 9                                      # Index position of maximum
of array

np.sqrt(arr)                                        # To calculate square root of all
elements in an array
arr.mean()                                          # To calculate mean of all the
values in an array
np.exp(arr)                                         # To calculate exponential value of
each element in an array


arr = np.arange(0,16)                               # Using reshape we can change the
dimensions of the array
arr_2D = arr.reshape(2,8)
arr_2D.flatten()                                     # Flatten is used to convert a 2D
array to 1D array
arr_2D.transpose()                                   # Transpose is used to convert the
rows into columns and vice-versa
```

Concatenate

```
arr_x = np.array([[1,2,3,4],[5,6,7,8]])               # Lets create 2 arrays
arr_y = np.array([[21,22,23,24],[25,26,27,28]])

IN: np.concatenate((arr_x, arr_y), axis=1)                # Join 2 arrays along columns
OUT: np.concatenate((arr_x, arr_y), axis=1)                # Join 2 arrays along columns
np.concatenate((arr_x, arr_y), axis=1)                # Join 2 arrays along columns
array([[ 1,  2,  3,  4, 21, 22, 23, 24],
       [ 5,  6,  7,  8, 25, 26, 27, 28]])

np.hsplit(arr, 2)                                  # It will split the array into 2
equal halves along the columns
np.vsplit(arr_z, 2)                                  # It will split the array into 2
equal halves along the rows
```

Takeaways
Using concatenate function we can merge arrays columnwise and rowwise. Also arrays can be horizontally and vertically spliited using hsplit and vsplit.

Conclusion
Numpy is open source add on module to Python.

By using NumPy you can speed up your workflow and interface with other packages in the Python ecosystem that use NumPy under the hood.
A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays.
It provide common mathematical and numerical routines in pre-compiled, fast functions.
It provides basic routines for manipulating large arrays and matrices of numeric data.
Key Features

NumPy arrays have a fixed size decided at the time of creation. Changing the size of an ndarray will create a new array and delete the original.
The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory.
NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data.

Create a random vector of size 12 , sort it in decreasing order, make 4*3 array using it.

```
import numpy as np
def generate():
    rand_vect = np.random.random(12)
    rand_sort = np.array(sorted(rand_vect,reverse=True))
    rand_new = rand_sort.reshape(4,3)
    print(rand_new)
generate()
```

Below is Python dictionary data and Python list labels. Create a DataFrame df from this dictionary data which has index as labels. Select the rows where the age is missing, i.e. is NaN

```
import numpy as np
data = {'animal': ['cat', 'cat', 'snake', 'dog', 'dog', 'cat', 'snake', 'cat', 'dog',
'dog'],
        'age': [2.5, 3, 0.5, np.nan, 5, 2, 4.5, np.nan, 7, 3],
        'visits': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
        'priority': ['yes', 'yes', 'no', 'yes', 'no', 'no', 'no', 'yes', 'no', 'no']}

labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
import pandas as pd
def generate():
    df = pd.DataFrame(data, index=labels)
    print(df[df['age'].isnull()])
generate()
```

From the give dataframe df, in the 'animal' column change the 'dog' entries to 'Labrador', change the age in row 'd' to 2.7 and calculate the mean age (name it mean_age) for each different animal in df. Print df and return mean_age

```
import pandas as pd
import numpy as np
data = {'animal': ['cat', 'cat', 'snake', 'dog', 'dog', 'cat', 'snake', 'cat', 'dog',
'dog'],
        'age': [2.5, 3, 0.5, np.nan, 5, 2, 4.5, np.nan, 7, 3],
        'visits': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
        'priority': ['yes', 'yes', 'no', 'yes', 'no', 'no', 'no', 'yes', 'no', 'no']}

labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

```
df = pd.DataFrame(data, index=labels)
def generate():
    df['animal'] = df['animal'].replace('dog', 'Labrador')
    df.loc['d', 'age'] = 2.7
    mean_age = df.groupby('animal')['age'].mean()
    print(df)
    return mean_age
generate()
```

Find the positions of numbers that are multiples of 4 from a series? Print series and return position. Hint: use argwhere.

```
import pandas as pd
import numpy as np
def generate():
    series = pd.Series(np.random.randint(1, 10, 7))
    pos=np.argwhere(series % 4 == 0 )
    print(series)
    print(pos)
generate()
```

Create an numpy array sequence named num_seq when only the starting point, step size and length of sequence is given

```
import numpy as np
length = 15
start = 5
step = 2

def generate(start, length, step):
    end = start + (step*length)
    num_seq = np.arange(start, end, step)
    return num_seq

generate(start, length, step)
```

reate a DatetimeIndex that contains each business day of 2016 and use it to index a Series of random numbers. Let's call this Series bus_2k16. Hint: use pd.date_range.

```
import pandas as pd
import numpy as np
def generate():
    dti = pd.date_range(start='2016-01-01', end='2016-12-31', freq='B')
    bus_2k16 = pd.Series(np.random.rand(len(dti)), index=dti)
    print(bus_2k16)
generate()
```

For each calendar month in dataframe bus_2k16, from previous problem. Print out the mean of values also find the sum of the values in bus_2k16 for every Monday.

```
import pandas as pd
import numpy as np
def generate():
    dti = pd.date_range(start='2016-01-01', end='2016-12-31', freq='B')
    bus_2k16 = pd.Series(np.random.rand(len(dti)), index=dti)
    print(bus_2k16)
    print(bus_2k16.resample('M').mean())
    print(bus_2k16[bus_2k16.index.weekday == 0].sum())
    return None
generate()
```

```
8x8 matrix 0 & 1 alternate

import numpy as np
def generate():
    Z = np.zeros((8,8),dtype=int)
    Z[1::2,::2] = 1
    Z[::2,1::2] = 1
    print(Z)
generate()
```

Pandas

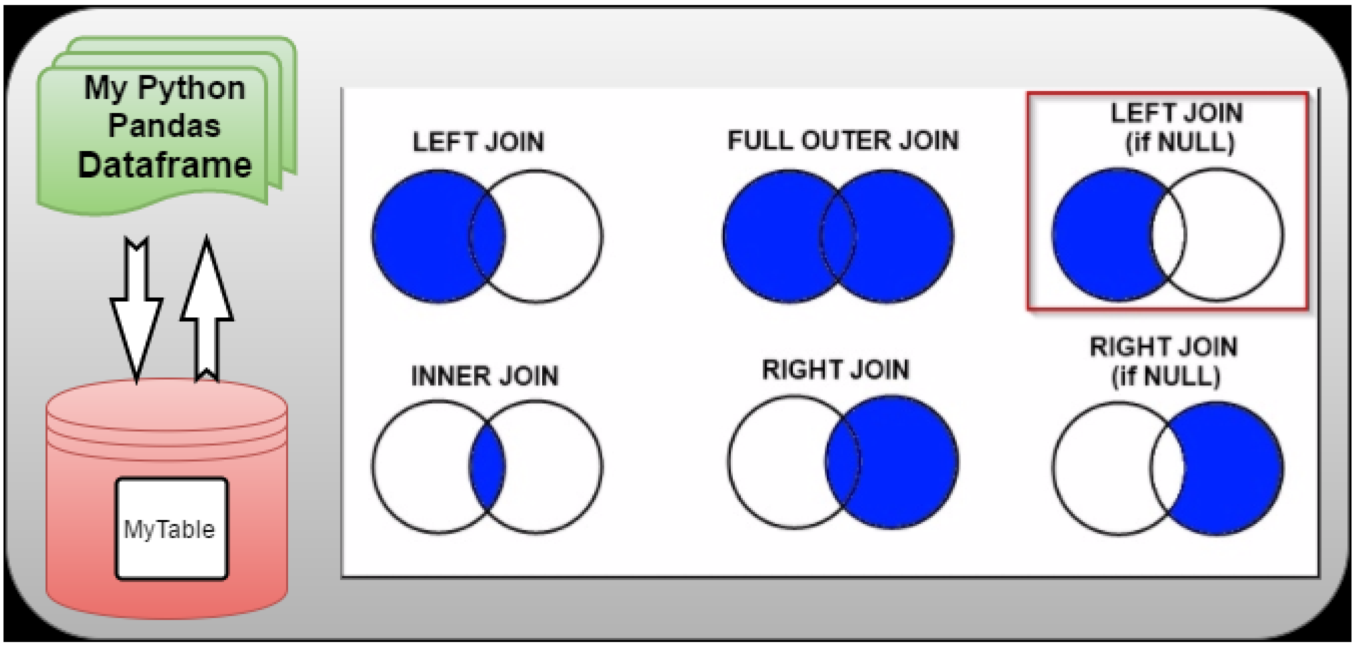
```
import pandas as pd
from IPython.display import display

csv_df = pd.read_csv("http://........")   #  # read_csv is used to read csv file
```

```
Create a dataframe 4x200 with random values

df = pd.DataFrame(columns=['COL_A','COL_B','COL_C','COL_D'], index=range(1,201))
#defining dataframe
df["COL_A"] = np.random.choice(["True", "False"], len(df))       #Creating random column
with True & False
df["COL_B"] = np.random.choice(["yes", "no"], len(df))           #Creating random column
with yes & no
df["COL_C"] = np.random.choice(["1", "0"], len(df))               #Creating random column
with 0 & 1
df['COL_D'] =  np.arange(len(df))                           # Adding a new column with
numbers 1-200
df["COL_X"] = np.random.randint(150,200, len(df)) *5       #Adding Radom numbers in range
- *5 is step size

df['Index'] =  np.arange(len(df))                           # Adding a new column with
numbers 1-200
df = df.set_index('Index')                                 #setting the index

df = pd.DataFrame({'COL_A':['True'],'COL_B':['yes'],'COL_C':['3']})    #Appending a row
on a DataFrame

df2 = df.append(df1, ignore_index = True)     #Appending a data frame by ignoring the
index
df2.reset_index(inplace=True)                #resetting the index
df2['index']                                #deleting the index
```

```
df.shape
df.count()
df.index
df.columns
df = df.set_index('C')          #setting the index
df['COL_A'].unique()               #Unique values
df['COL_A'].nunique()              #count of Unique values
df['COL_A']..value_counts()         # Observe the number of counts


df[3:7]                                               # Accessing/Filtering all the
elements from 3rd to 7th index
df['COL_A']                                                # Accessing the data using
labels
df[['COL_A','COL_D']]                                            # Accessing the data
using multiple labels
df[['1','2','3']]                                      # Accessing multiple elements in
a series using labels of dict
df['COL_E'] = df['COL_C'] + df['COL_D']                          # Addition of two columns.
You can perform any math operation.


df.drop('COL_E', axis=1, inplace=False)             # Drop or modify the dataframe
columns
df.drop("COL_E", axis=0, inplace =False)             # Drop or modify the dataframe row
after setting index
df.drop(['COL_E'], axis=1)

df.drop(df.index[1:5], axis=0, inplace =True)      # Drop or modify the dataframe rows
after setting index
del df['COL_E']                                       # Column Deletion using del

- Indexing using iloc
df.iloc[0:4,1:3]                                   # It will return rows from 0-4 and
columns from 0-5
df.iloc[:,0:5]                                     # Return the columns from 0-5
df.iloc[0:4,:]                                     # Return all the rows from 0-4

df.loc[3:5,"COL_C"]                                  # Returns index "D" 1-4 and column
"C"
df.loc[1:4,"COL_B":"COL_C"]                           # Returns index "D" 1-4 and columns
B to C
df.loc[:,"COL_B":]                                 # Return all the columns from "B"
onwards and all the rows
```

```
Merging, Concatenating and Appending
In the previous section we saw how to add rows or columns.
Here we will see how to merge two dataframes.

df1 = pd.DataFrame({                                      # Lets create 2 dataframes
    'id':[1,2,3,4,5],
    'name':['a','b','c','d','e'],
    'sub':['sub1','sub2','sub3','sub4','sub5']
})
df2 = pd.DataFrame({
    'id':[1,2,3,4,5],
    'name':['b','c','d','e','f'],
    'sub':['sub3','sub4','sub5','sub6','sub7']
})


pd.concat([df1, df2], axis=0)                             # Joining two DataFrame along
the rows
```

```
pd.concat([df1, df2], axis=1)                              # Joining 2 DataFrame along
the columns
pd.merge(left=df1, right=df2, on='sub')                    # Joining 2 DataFrame using
'sub' as keypd.merge(left=df1, right=df2, on='sub',how='left')
# on left
pd.merge(left=df1, right=df2, on='sub', how='outer')        # left, right, outer

another sample
IN:
Mylist1 = [(1,10),(2,20),(3,30),(4,40),(5,50)]
labels1 = ['ID','NUM']
df1_x = pd.DataFrame.from_records(Mylist1,columns= labels1)
df1_x
Mylist2 = [(1,'COL_A'),(6,'COL_B'),(3,'COL_C'),(8,'COL_D'),(10,'COL_E')]
labels2 = ['ID','ALPHA']
df2_x = pd.DataFrame.from_records(Mylist2,columns= labels2)
print(df1_x)
print(df2_x)

df_onlyleft = pd.merge(left=df1_x,right=df2_x,on='ID',how='left',indicator=
True).query('_merge=="left_only"').drop('_merge',1)
df_onlyleft

OUT
   ID  NUM
0   1   10
1   2   20
2   3   30
3   4   40
4   5   50
   ID ALPHA
0   1     A
1   6     B
2   3     C
3   8     D
4  10     E
ID  NUM ALPHA
1   2   20  NaN
3   4   40  NaN
4   5   50  NaN


IN:
df_left = df1_x
df_right = df2_x
df_onlyleft2 = df_left.query('ID not in @df_right.ID')
df_onlyleft2

OUT:
ID  NUM
1   2   20
3   4   40
4   5   50
```



```
SORT & Filter

df['COL_D'] >=15                                          # This returns a Boolean Series
df[~(df['COL_D'] >=15)]                                     # This returns all the rows
for which the condition is True

df1 = df[(df['COL_D'] >=15) & (df['COL_B'] != 'yes')]
df1 = df[(df.COL_B =='yes')]['COL_D'].count()                      #Sort and count
df1 = df[(df.COL_B !='yes')]['COL_D'].Sum()                       #Sort and sum
df1 = df.groupby(['COL_A','COL_B','COL_C'])['COL_D'].agg(['sum','mean','count'])
#aggregate


df1 = df.groupby(['COL_A','COL_B'])        #group A & B and count / mean /sum
df1.count()
df1.mean()
df1.sum()


df1 = df.groupby('COL_A').sum()      #group only values of A  and sum


df1 = df.set_index(['COL_A','COL_B'])                                   # Lets set 'sex'
and 'size' as our index
df1.sort_index(inplace = True)                                   # Sorting all the index
of A & B values in ascending order
df1 = df1.loc['true']                                                       # Accessing
records if index is A
df2 = df1.xs('yes', level='COL_B')                                 # Accessing records if
serving size is 2 (From second index)

df.sort_values('COL_C')                                               # Sorting the
records based on a column
```

```
Apply function

def times2(x):                                                                # We are
going to apply this function
    num = x
    if x%6==0:
        num *= 2
    return num

df1 = df['COL_D'].apply(times2)

OR

df['COL_D'] = df['COL_D'].apply(lambda x: x * 2)
df
```

```
# DATE & time
df['Date'] = pd.to_datetime(df['Date'], format='%m/%d/%Y')
df['Time'] = pd.to_datetime(df['Time'], format = '%H:%M:%S')

year = df.Date.dt.year                               # Extracting Year from Date column
print(year.head())

month = df.Date.dt.month                             # Extracting Month from Date column
print(month.head())

day = df.Date.dt.day                                 # Extracting Day from Date column
print(day.head())

day_of_week = air_df.Date.dt.dayofweek                    # Extracting the day of the week
number
print(day_of_week.head())

day_name = air_df.Date.dt.weekday_name                    # Extracting the name of the day
print(day_name.head())

day_of_year = air_df.Date.dt.dayofyear                    # Extracting the day of the year
print(day_of_year.head())

hour = air_df.Time.dt.hour                                # Extracting the hour from time
print(hour.head())

minute = air_df.Time.dt.minute                            # Extracting the minutes from the
time
print(minute.head())

second = air_df.Time.dt.second                            # Extracting the seconds from the
time
print(second.head())

 measure the number of records before 01/01/2005
datestamp = pd.to_datetime("01/01/2005", format='%d/%m/%Y')

from datetime import timedelta
df1 = df[air_df['Date'] < datestamp].tail()
```

```
Convert the first character of each element in a series to uppercase?And return the new
series. Hint: use map and lambda methods

import pandas as pd
series = pd.Series(['how', 'to', 'learn', 'data science?'])
def generate():
    new_series = series.map(lambda x: x[0].upper() + x[1:])
    return new_series
generate()
```

```
Given the dataframe df, In the From_To column split each string on the underscore
delimiter _ to give a new temporary DataFrame with the correct values. Also standardise
the strings so that only the first letter is uppercase (e.g. "londON" should become
"London".) Assign the correct column names to this temporary DataFrame named as temp.
Return temp

import numpy as np
df = pd.DataFrame({'From_To': ['LoNDon_paris', 'MAdrid_miLAN', 'londON_StockhOlm',
                               'Budapest_PaRis', 'Brussels_londOn'],
              'FlightNumber': [10045, np.nan, 10065, np.nan, 10085],
              'RecentDelays': [[23, 47], [], [24, 43, 87], [13], [67, 32]],
                   'Airline': ['KLM(!)', '<Air France> (12)', '(British Airways. )',
                               '12. Air France', '"Swiss Air"']})
import pandas as pd
def generate():
    temp = df.From_To.str.split('_', expand=True)
    temp.columns = ['From', 'To']
    temp['From'] = temp['From'].str.capitalize()
    temp['To'] = temp['To'].str.capitalize()
    return temp
generate()
```

```
Given a dataFrame df with a column 'int' of integers. Print out the dataframe with
filtered out rows which contain the same integer as the row immediately above. Also print
out the dataframe after subtracting the mean of all integers from each element.

import pandas as pd
df = pd.DataFrame({'int': [1, 2, 2, 3, 4, 5, 5, 5, 6, 7, 7]})
def generate():
    print(df.drop_duplicates(subset='int'))
    print(df.sub(df.mean(axis=0), axis=1))
    return None
generate()
```

```
Filter out and capitalise all letters of words that contain atleast 2 vowels from a
series?

import pandas as pd
series = pd.Series(['Insaid', 'strives', 'for', 'bringing','out','the', 'best','in','you'
])
def generate():
    from collections import Counter
    mask = series.map(lambda x: sum([Counter(x.lower()).get(i, 0) for i in
list('aeiou')]) >= 2)
    s = series[mask].str.upper()
    print(s)
    return None
generate()
```

```
df.columns = map(str.lower, df.columns)          #mapping all headers to lower case
df.dtypes
```

```
#adding zeros in missing data Nan Values

df.c1.fillna('0', inplace=True)
df.c2.fillna('0', inplace=True)
```

```
Removing string prefix

df.c1 = cd.c1.loc[:,].replace(regex=True, to_replace="abc", value="")

df.c2 = df.c2.loc[:,].replace(regex=True, to_replace=("01. ","02. ","03. ","04. ","05.
","06. ","07. ","08. ","09. ","10. ","11. ","12. ","05A. ","05B. "), value="")

TO remove suffix from entire column
df.c2 = df.c2.loc[,:].replaceregex=True, to_replace=("___ ","___"), value="")
```

```
cols = ['c1', 'c2','c3']        #converting Float to integer
df[cols] = df[cols].applymap(np.int64)
```

```
df1 = df.copy(deep=True)                    #creating copy of dset
```

```
nlargest
df.nlargest(20,'c1').head(20)   #top20
```

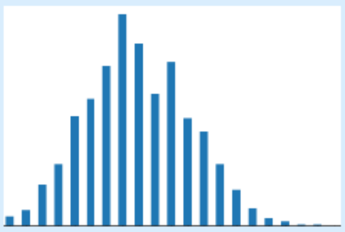
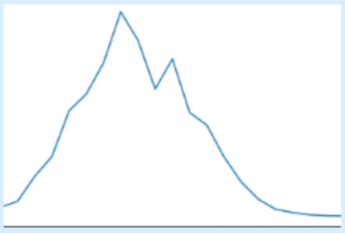
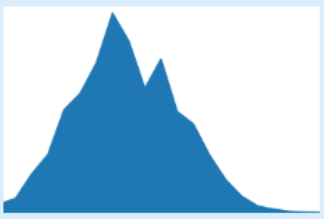
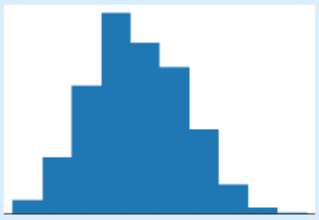
```
totals = df[['c1','c2','c3']]   #totals of selected columns
totals.sum(axis = 0)

#group total and sort by year
total_each_year = df.groupby(['year',])['XYZ'].sum().reset_index()
#grouping
total_each_year = total_each_year.sort_values(by = ['XYZ','year'], ascending=False)
    #sorting
total_each_year
```

```
df['percent'] = df.groupby('c1').apply(lambda s: s.C1.nunique()/s.C1).values
```

Data Visualization

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib as mat
import bokeh
```

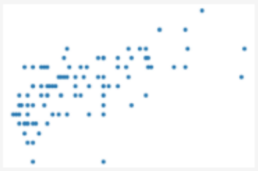
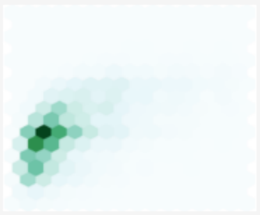
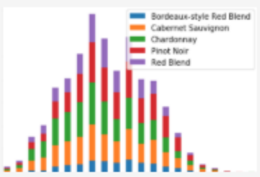
Univariate Plotting with pandas

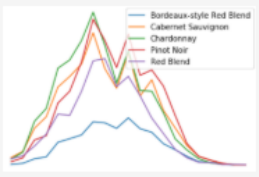





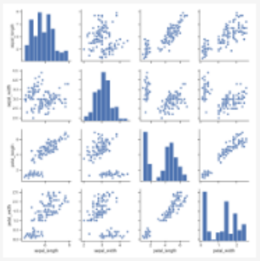
Bar Chat	Line Chart	Area Chart	Histogram
df.plot.bar()	df.plot.line()	df.plot.area()	df.plot.hist()
Good for nominal and small ordinal categorical data.	Good for ordinal categorical and interval data.	Good for ordinal categorical and interval data.	Good for interval data.

```
Univariate Plotting with pandas

Bar charts

df['X'].value_counts().head(10).plot.bar()
(df['X'].value_counts().head(10) / len(df)).plot.bar()
df['X'].value_counts().sort_index().plot.bar()

Line charts
df['X'].value_counts().sort_index().plot.line()

Area Charts
df['X'].value_counts().sort_index().plot.area()

Histogram
df[df['X'] < 200]['X'].plot.hist()
```

Bivariate plotting with pandas



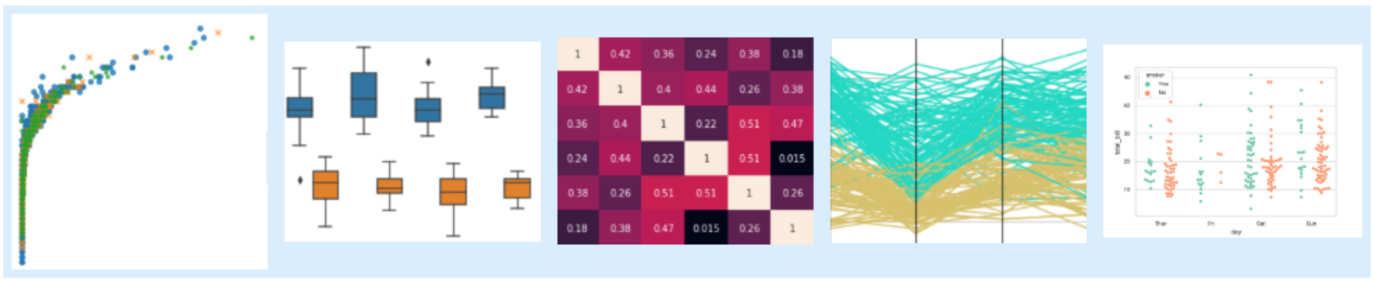


Scatter Plot	Hex Plot	Stacked Bar Chart	Bivariate Line Chart	Pair Plot
df.plot.scatter()	df.plot.hex()	df.plot.bar(stacked=True)	df.plot.line()	df.sns.pairplot()
Good for interval and some nominal categorical data.	Good for interval and some nominal categorical data.	Good for nominal and ordinal categorical data.	Good for ordinal categorical and interval data.	Good for finding pairwise relationship in the data.

```
Bivariate plotting with pandas

Many pandas multivariate plots expect input data to be in this format, with:
one categorical variable in the columns
one categorical variable in the rows
counts of their intersections in the entries.

Scatter plot
df[df['X'] < 100].sample(100).plot.scatter(x='X', y='Y')

Hexplot
df[df['X'] < 100].sample(100).plot.hexbin(x='X', y='Y', gridsize=12)

Stacked plots
```

```
df.plot.bar(stacked=True, figsize=(20,8))

plot1 = df.groupby(["A", "B"])['C'].sum().unstack('B').fillna(0)
plot1.plot(kind='bar', stacked=True,figsize=(20,8) )

Area
df.plot.area()

Line
df.plot.line()

Pair Plot
sns.pairplot(df.drop("X", axis=1), hue="Y",palette="viridis", size=2)
sns.pairplot(df.drop("X", axis=1), hue="Y",palette="inferno", size=2)
```

Multivariate plotting



Multivariate Scatter Plot	Grouped Box Plot	Heatmap	Parallel Coordinates	Swarm plot
df.plot.scatter()	df.plot.box()	sns.heatmap	pd.plotting.parallel_coordinates	sns.swarmplot()
Good for interval and some nominal categorical data.	Good for interval and some nominal categorical data.	Good for nominal and ordinal categorical data.	Good for ordinal categorical and interval data.	Good for drawing categorical variable with non-overlapping points.

```
Multivariate scatter plots

sns.lmplot(x='X', y='Y', hue='ZZZZ', markers=['o', 'x', '*']
           data=df.loc[df['ZZZZ'].isin(['col_val', 'col_val', 'col_val'])],
           fit_reg=False)


Box Plot
sns.boxplot(x="X", y="Y", hue='ZZZZ', data=df)

heatmap
sns.heatmap(df, annot=True)

Parallel Coordinates
parallel_coordinates(df, 'X')

Swarm Plot
sns.swarmplot(x="x", y="Y", hue="ZZZZ", palette="gnuplot", data=df)
```

```
from bokeh.plotting import Figure,figure, output_file, show, output_notebook
from bokeh.layouts import column
from bokeh.models import ColumnDataSource, CustomJS, Slider, HoverTool
output_notebook()

plot = figure(plot_width=300, plot_height=300)
plot.circle(x=[1,2,3], y=[4,5,6], size=20,
                  color="#FB8072", fill_alpha=0.2, line_width=2)
slider = Slider(start=.1, end=1., value=.2, step=.1, title="delta-V")
```

```
show(plot)
```

```
CATPLOT

sns.catplot("C1", "C2", data=df, kind="bar" ,palette="PuBuGn_d",height=6, aspect=2)
plt.xlabel('C1')
plt.ylabel('C2')
locs, labels = plt.xticks()
plt.setp(labels, rotation=55)
plt.show()
```

```
# Distribution using Facetgrid

as_fig = sns.FacetGrid(df,hue='C1',aspect=5)

as_fig.map(sns.kdeplot,'C2',shade=True)

ABC = df['C2'].max()

as_fig.set(xlim=(0,ABC))

as_fig.add_legend()
plt.title('NAME_____')
```

Linear Regression

Linear regression is a *basic* and *commonly* used type of **predictive analysis**. The overall idea of regression is to examine two things:

- Does a set of **predictor variables** do a good job in predicting an **outcome** (dependent) variable?
- Which variables in particular are **significant predictors** of the outcome variable, and in what way they do **impact** the outcome variable?

These regression estimates are used to explain the **relationship between one dependent variable and one or more independent variables**. The simplest form of the regression equation with one dependent and one independent variable is defined by the formula :
$y = \beta_0 + \beta_1 x$

What does each term represent?

- y is the response
- x is the feature
- β_0 is the intercept
- β_1 is the coefficient for x

Clean the data before applying any ML Algo

```
1. Write a code to understand the total count and percentage of missing values.

def missing_data(data):
    total = data.isnull().sum().sort_values(ascending = False)
    percent = (data.isnull().sum()/data.isnull().count()*100).sort_values(ascending = 
False)
    return pd.concat([total, percent], axis=1, keys=['Total', 'Percent'])
missing_data(data)


2. Impute the missing value using "median" groupby Column1.

def Miss():
    data["Column_missing_values"].fillna(data.groupby("Column1")
["Column_missing_values"].transform("median"), inplace=True)
    return
Miss()
print (data.isnull().sum())


3.  Plot outliers

def plotoutliers():
    import seaborn as sns
    sns.catplot(data="____", orient="h", palette="Set2", kind="box", height=6, aspect=2)
    return None
plotoutliers()



4.  Fix outliers

def outliers(data):
    import pandas as pd
    Q1 = data.quantile(0.05)
    Q3 = data.quantile(0.95)
    Q_diff = Q3 - Q1
    XYZ = data[~((data < (Q1 - Q_diff))|(data > (Q3 + Q_diff))).any(axis=1)]
    print(data[((data < (Q1 - Q_diff))|(data > (Q3 + Q_diff))).any(axis=1)])
    return XYZ
data = outliers(data)
data
```

```
Assumption validation before starting

****MAKE SURE YOUR TARGET VALRIABLE IS CONTINIOUS*******

1. There should be a linear and additive relationship between dependent (response)
variable and independent (predictor) variable(s). A linear relationship suggests that a
change in response Y due to one unit change in X¹ is constant, regardless of the value of
X¹. An additive relationship suggests that the effect of X¹ on Y is independent of other
variables.

    CHECK WITH
#sns.pairplot(DATASET, size = 2, aspect = 1.5)
```

```
2. There should be no correlation between the residual (error) terms. Absence of this
phenomenon is known as Autocorrelation.
3. The independent variables should not be correlated. Absence of this phenomenon is
known as multicollinearity.

    CHECK WITH
DATASET.corr()

OR

sns.heatmap(DATASET.corr(), annot=True );

4. The error terms must have constant variance. This phenomenon is known as
homoskedasticity. The presence of non-constant variance is referred to
heteroskedasticity.

     CHECK WITH
sns.pairplot(DATASET, x_vars=['EV1', 'EV2', 'EV3'], y_vars='TV', size=5, aspect=1,
kind='reg')

OR

JG1 = sns.jointplot("EV1", "TV", data=DATASET, kind='reg')       #EV = EXPLAINATORY
VARIABLE
JG2 = sns.jointplot("EV2", "TV", data=DATASET, kind='reg')       #TV = TARGET VARIABLE
JG3 = sns.jointplot("EV3", "TV", data=DATASET, kind='reg')
#subplots migration
f = plt.figure()
for J in [JG1, JG2,JG3]:
    for A in J.fig.axes:
        f._axstack.add(f._make_key(A), A)

5. The error terms must be normally distributed.

   CHECK WITH
f, axes = plt.subplots(2, 2, figsize=(7, 7), sharex=True)
     # Set up the matplotlib figure
sns.despine(left=True)

sns.distplot(DATASET.TV, color="b", ax=axes[0, 0])

sns.distplot(DATASET.EV1, color="r", ax=axes[0, 1])

sns.distplot(DATASET.EV2, color="g", ax=axes[1, 0])

sns.distplot(DATASET.EV3, color="m", ax=axes[1, 1])
```

Linear Regression starts

1. Standardize features by removing the mean and scaling to unit standard deviation.

#EV = EXPLAINATORY VARIABLE

#TV = TARGET VARIABLE

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler().fit(DATASET)
DATASET_R = scaler.transform(DATASET)
```

```
DATASET_R.head()
```

2. adding back the columns names

```
DATASET_R.columns = ['TV','EV1','EV2','EV3']
DATASET_R.head()
```

3. Preparing X and y

```
feature_cols = ['EV1', 'EV2', 'EV3']                # create a Python list of feature
names
X = DATASET_R[feature_cols]                         # use the list to select a subset of
the original DataFrame-+
y = DATASET_R.TV
```

4. Splitting X and y into training and test datasets.

```
from sklearn.model_selection import train_test_split

def split(X,y):
    return train_test_split(X, y, test_size=0.20, random_state=1)

X_train, X_test, y_train, y_test=split(X,y)
print('Train cases as below')
print('X_train shape: ',X_train.shape)
print('y_train shape: ',y_train.shape)
print('\nTest cases as below')
print('X_test shape: ',X_test.shape)
print('y_test shape: ',y_test.shape)
```

5. Linear regression in scikit-learn

To apply any machine learning algorithm on your dataset, basically there are 4 steps:

1. Load the algorithm
2. Instantiate and Fit the model to the training dataset
3. Prediction on the test set
4. Calculating Root mean square error The code block given below shows how these steps are carried out:

```
from sklearn.linear_model import LinearRegression
   linreg = LinearRegression()
   linreg.fit(X_train, y_train)
   RMSE_test = np.sqrt(metrics.mean_squared_error(y_test, y_pred_test))
```

```
def linear_reg( X, y, gridsearch = False):

    X_train, X_test, y_train, y_test = split(X,y)

    from sklearn.linear_model import LinearRegression
    linreg = LinearRegression()

    if not(gridsearch):
        linreg.fit(X_train, y_train)

    else:
        from sklearn.model_selection import GridSearchCV
        parameters = {'normalize':[True,False], 'copy_X':[True, False]}
        linreg = GridSearchCV(linreg,parameters, cv = 10,refit = True)
        linreg.fit(X_train, y_train)                            # fit the model to the 
training data (learn the coefficients)
        print("Mean cross-validated score of the best_estimator : ", linreg.best_score_)


    y_pred_test = linreg.predict(X_test)                                      
  # make predictions on the testing set

    RMSE_test = (metrics.mean_squared_error(y_test, y_pred_test))                            
# compute the RMSE of our predictions
    print('RMSE for the test set is {}'.format(RMSE_test))

    return linreg
```

6. Checking the RMSE

```
X = DATASET_R[feature_cols]
y = DATASET_R.TV
linreg = linear_reg(X,y)
```

7. Interpreting Model Coefficients

```
feature_cols.insert(0,'Intercept')
coef = linreg.coef_.tolist()
coef.insert(0, linreg.intercept_)
eq1 = zip(feature_cols, coef)
for c1,c2 in eq1:
    print(c1,c2)
```

Write your equation

y(TV) = 0.00116 + 0.7708 * EV1 + 0.508 * EV2 + 0.010 * EV3

Make initial interpretation 1. 2.

Important Notes:

- This is a statement of **association**, not **causation**.
- E.g. If an increase in television ad spending was associated with a **decrease** in sales, β1 would be **negative.**

```
8. Using the Model for Prediction

y_pred_train = linreg.predict(X_train)
y_pred_test = linreg.predict(X_test)
print(y_pred_train)
print(y_pred_test)

#We need an evaluation metric in order to compare our predictions with the actual values.
```

9. Model evaluation

Error is the *deviation* of the values *predicted* by the model with the *true* values.
For example, if a model predicts that the price of apple is Rs75/kg, but the actual price of apple is Rs100/kg, then the error in prediction will be Rs25/kg.
Below are the types of error we will be calculating for our *linear regression model*:

- Mean Absolute Error
- Mean Squared Error
- Root Mean Squared Error

9.1 Model Evaluation using metrics.

Mean Absolute Error (MAE) is the mean of the absolute value of the errors:

$$\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Computing the MAE for our TV predictions

```
MAE_train = metrics.mean_absolute_error(y_train, y_pred_train)
MAE_test = metrics.mean_absolute_error(y_test, y_pred_test)
print('MAE for training set is {}'.format(MAE_train))
print('MAE for test set is {}'.format(MAE_test))
```

9.2 Mean Squared Error (MSE) is the mean of the squared errors:

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Computing the MSE for our TV predictions

```
MSE_train = metrics.mean_squared_error(y_train, y_pred_train)
MSE_test = metrics.mean_squared_error(y_test, y_pred_test)
print('MSE for training set is {}'.format(MSE_train))
print('MSE for test set is {}'.format(MSE_test))
```

9.3 Root Mean Squared Error (RMSE) is the square root of the mean of the squared errors:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Computing the RMSE for our TV predictions

```
RMSE_train = np.sqrt( metrics.mean_squared_error(y_train, y_pred_train))
RMSE_test = np.sqrt(metrics.mean_squared_error(y_test, y_pred_test))
print('RMSE for training set is {}'.format(RMSE_train))
print('RMSE for test set is {}'.format(RMSE_test))
```

Comparing these metrics:

- **MAE** is the easiest to understand, because it's the **average error.**
- **MSE** is more popular than MAE, because MSE "punishes" larger errors.

- **RMSE** is even more popular than MSE, because RMSE is *interpretable* in the "y" units.
  - Easier to put in context as it's the same units as our response variable.

10. Model Evaluation using Rsquared value.

- There is one more method to evaluate linear regression model and that is by using the **Rsquared** value.
- R-squared is the **proportion of variance explained**, meaning the proportion of variance in the observed data that is explained by the model, or the reduction in error over the **null model**. (The null model just predicts the mean of the observed response, and thus it has an intercept and no slope.)
- R-squared is between 0 and 1, and higher is better because it means that more variance is explained by the model. But there is one shortcoming of Rsquare method and that is **R-squared will always increase as you add more features to the model**, even if they are unrelated to the response. Thus, selecting the model with the highest R-squared is not a reliable approach for choosing the best linear model.

There is alternative to R-squared called **adjusted R-squared** that penalizes model complexity (to control for overfitting).

```
yhat = linreg.predict(X_train)
SS_Residual = sum((y_train-yhat)**2)
SS_Total = sum((y_train-np.mean(y_train))**2)
r_squared = 1 - (float(SS_Residual))/SS_Total
adjusted_r_squared = 1 - (1-r_squared)*(len(y_train)-1)/(len(y_train)-X_train.shape[1]-1)
print(r_squared, adjusted_r_squared)
```

```
yhat = linreg.predict(X_test)
SS_Residual = sum((y_test-yhat)**2)
SS_Total = sum((y_test-np.mean(y_test))**2)
r_squared = 1 - (float(SS_Residual))/SS_Total
adjusted_r_squared = 1 - (1-r_squared)*(len(y_test)-1)/(len(y_test)-X_test.shape[1]-1)
print(r_squared, adjusted_r_squared)
```

Whereas R-squared is a relative measure of fit, RMSE is an absolute measure of fit. As the square root of a variance, RMSE can be interpreted as the standard deviation of the unexplained variance, and has the useful property of being in the same units as the response variable. Lower values of RMSE indicate better fit. & R-square should be high

11. Feature Selection

At times some features do not contribute much to the accuracy of the model, in that case its better to discard those features.

- Let's check whether **dropping one of the EV** improve the quality of our predictions or not.
  To check this we are going to take all the features other than "EV" and see if the error (RMSE) is reducing or not.
- Also Applying **Gridsearch** method for exhaustive search over specified parameter values of estimator.

```
feature_cols = ['EV1','EV2']                # create a Python list of feature names AND
DROPPING EV3
X = DATASET_R[feature_cols]
y = DATASET_R.sales
linreg=linear_reg(X,y, gridsearch=True)
```

12. Linear Regression model with GridSearchCV

```
feature_cols = ['TV','EV1']                                                                    #
create a Python list of feature names
X = DATASET_R[feature_cols]
y = DATASET_R.TV
linreg=linear_reg(X,y, gridsearch=True)
```

Note:

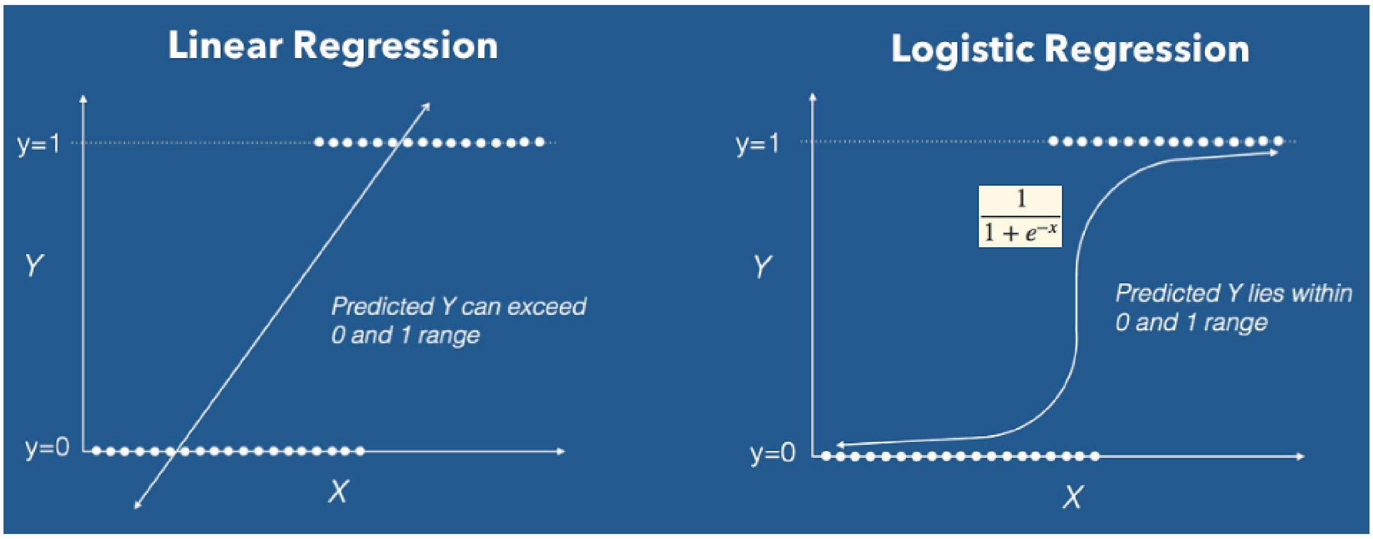
If there are categorical variables present use ONE HOT ENCODING - dummyfication ; Make sure you drop one of the feature to avoid dummy variable trap or multicollinearity

Logistic Regression

Logistic regression is a techinque used for solving the **classification problem**.
And Classification is nothing but a problem of **identifing** to which of a set of **categories** a new observation belongs, on the basis of *training dataset* containing observations (or instances) whose categorical membership is known.
For example to predict:
Whether an email is spam (1) or not (0) or,
Whether the tumor is malignant (1) or not (0)
Below is the pictorial representation of a basic logistic regression model to classify set of images into *happy or sad.*

Both Linear regression and Logistic regression are **supervised learning techinques**. But for the *Regression* problem the output is **continuous** unlike the *classification* problem where the output is **discrete**.

- Logistic Regression is used when the **dependent variable(target) is categorical**.
- **Sigmoid function** or logistic function is used as *hypothesis function* for logistic regression. Below is a figure showing the difference between linear regression and logistic regression, Also notice that logistic regression produces a logistic curve, which is limited to values between 0 and 1.


Linear Regression
y=1
Y
y=0
X
Predicted Y can exceed 0 and 1 range
Logistic Regression
y=1
$\frac{1}{1+e^{-x}}$
Y
y=0
X
Predicted Y lies within 0 and 1 range

- Clean Data Before you start

ASSUMPTIONS OF LOGISTIC REGRESSION

1. ASSUMPTION OF APPROPRIATE OUTCOME STRUCTURE To begin, one of the main assumptions of logistic regression is the appropriate structure of the outcome variable. Binary logistic regression requires the dependent variable to be binary and ordinal logistic regression requires the dependent variable to be ordinal.

2. ASSUMPTION OF OBSERVATION INDEPENDENCE Logistic regression requires the observations to be independent of each other. In other words, the observations should not come from repeated measurements or matched data. ASSUMPTION OF THE ABSENCE OF MULTICOLLINEARITY Logistic regression requires there to be little or no multicollinearity among the independent variables. This means that the independent variables should not be too highly correlated with each other.

3. ASSUMPTION OF LINEARITY OF INDEPENDENT VARIABLES AND LOG ODDS Logistic regression assumes linearity of independent variables and log odds. Although this analysis does not require the dependent and independent variables to be related linearly, it requires that the independent variables are linearly related to the log odds.

4. ASSUMPTION OF A LARGE SAMPLE SIZE Finally, logistic regression typically requires a large sample size. A general guideline is that you need at minimum of 10 cases with the least frequent outcome for each independent variable in your model. For example, if you have 5 independent variables and the expected probability of your least frequent outcome is .10, then you would need a minimum sample size of 500 (10*5 / .10).

Logistic regression is quite different than linear regression in that it does not make several of the key assumptions that linear and general linear models (as well as other ordinary least squares algorithm based models) hold so close:

- Logistic regression does not require a linear relationship between the dependent and independent variables,
- The error terms (residuals) do not need to be normally distributed,
- Homoscedasticity is not required, and
- The dependent variable in logistic regression is not measured on an interval or ratio scale.

_*source* # https://www.lexjansen.com/wuss/2018/130_Final_Paper_PDF.pdf (https://www.lexjansen.com/wuss/2018/130_Final_Paper_PDF.pdf)