Contents

[Set{}: 2](#_Toc141273699)

[list[]: 2](#_Toc141273700)

[Tuple(): 3](#_Toc141273701)

[dictionary{key:value}: 4](#_Toc141273702)

[Numpy: 4](#_Toc141273703)

[Pandas: 5](#_Toc141273704)

[Matplotlib(Line plot): 5](#_Toc141273705)

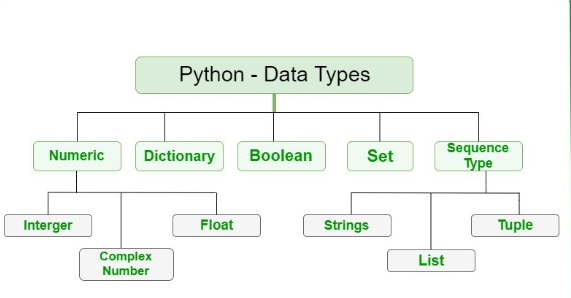
[Matplotlib(scatter, bar, stacked, histogram): 6](#_Toc141273706)

[Global, local and nonlocal 6](#_Toc141273707)

[Passing value to inner function by calling outer function 7](#_Toc141273708)

[Flexible arguments(\*args,\*\*kwargs): 8](#_Toc141273709)

[Lambda function: 9](#_Toc141273710)



# Set{}:

Assignment: set={1,4,43}

* Data will be stored in hash order
* We cannot assign or retrieve value like **set[0]=1** or **print(set[1])**
* Set to kunai index ko value access garnu paryo vaney write:  
  list(set\_name)[index\_value]: typecast set to list and then access items
* Sets do not maintain the order in which elements were added. When you add an element to a set, Python hashes the element to compute its hash value. This hash value is then used to determine the index where the element should be stored in the underlying array.
* If want to achieve the order of elements then we use **ordered.dict** or **ordered.set**

see values like:  
set={1,4,43}

for x in  set:

    print(x)

# list[]:

Assignment: list=[2,1]

* Data will be stored in the predefined order
* We can assign value or retrieve like list[0]=1 or print(list[0])
* Negative indexing is possible.
* List slicing is also possible:  
  list[0:4] : items from index 0 to 3 making a total of 4 items  
  list[1:]: all items from index 1  
  list[:4] all items from index 0 to 3  
  list[-1]: last item, list[-2] : 2nd last item
* Len(list): size of list
* string = input("Enter elements (Space-Separated): ")    
  # split the strings and store it to a list  
  lst = string.split()
* List.remove(5): removes item 5 from the list  
  List.pop(2) or del(List(2)) : removes from index 2
* Directly assigning to another list copies the reference rather than the value

lis1=[1,2,3]

lis2=lis1

print(lis1,lis2)

lis2[2]=4

print(lis1,lis2) #lis 1 is also change, as the reference is passed.

* Change nahula lai we should assign like:

Lis2=list(lis1)

Or

Lis2=lis1[:]

# Tuple():

* Immutable but can retrieve data through indexing as:  
  print(tup[1])

mytuple = (1, 2, 3, 4, 5)

# tuples are indexed

print(mytuple[1])

print(mytuple[4])

# tuples contain duplicate elements

mytuple = (1, 2, 3, 4, 2, 3)

print(mytuple)

We can assign tuple values to individual variable like

mytuple = (1, 2, 3)

a,b,c=mytuple

print(a)   #prints 1

print(b)   #prints 2

print(c)   #prints 3

# dictionary{key:value}:

* Value can be repeated but keys are immutable(can be deleted) and cannot be repeated
* Nested dictionary elements can be accessed like multidimensional array, where keys in place of index.
* Deleting: del dict\_name[‘key’]

# Numpy:

As a data scientist we want to do calculation on list but we want it to be fast. So we use Numpy.

* Convert list to numpy array:  
    
  weight=np.array(weight\_list)  
  height=np.array(height\_list)  
  bmi=weight/height\*\*2 : each respective array index mah perform huncha from both arrays
* Numpy arrays contain only one type. If a list having multiple types are converted to numpy arrays then it will be transferred all as string
* lis1=[1,2,3]

print(lis1+lis1) #[1,2,3,1,2,3]

lis\_arr=np.array(lsi1)

print(lis\_arr+lis\_arr) #[2,4,6]

* n\_array.shape : gives the structure of the array  
  n\_array[1][2] or n\_array[1,2] : does the same work of getting the element listed  
  n\_array[:,1:5]: gives all rows with 2nd to 4th column of all rows
* np.mean(),np.meadian(n\_Arr[ : , 1 : 24 ]), np.std(), np.round(), np.random.normal()

# Heights of the goalkeepers: gk\_heights

gk\_heights=np\_heights[np\_positions=='GK'] #np\_positions is array

# Heights of the other players: other\_heights

other\_heights=np\_heights[np\_positions!='GK']

import numpy as np

arr=np.array([1,2,3])

print(arr>1)

print(arr[arr>1])

output:

[False  True  True]

[2 3]

# Pandas:

* import pandas as pd
* read\_csv(path,header=None,usecols=[tyo columns haru matrai dekha], index\_col[tyo col lai index man]): reads csv from a location to a dataframe
* head(): list top 5 rows
* info(): explain the structure of the csv file

Column Operation in dataframe

* name.col\_name.sum()= gives sum of the column
* new\_variable = name.col\_name or new\_variable=name[‘col\_name’] : selects the value of a column In the new variable

Note:: Do not use .col\_name when there is a wildcard in the name of the column.

Rows operation in dataframe

* we can do operation in pandas’ column similarly as numpy array  
  np\_arr>5 (this will result in [false,true…true]) similar in pandas as well
* name[name.price>20]: selects only those rows having price greater than 20 (similar like numpy)

# Matplotlib(Line plot):

Import matplotlib as plt | **from matplotlib import pyplot as plt**

* plt.plot(x\_values,y\_values)
* plt.plot(x\_values,y\_values,label=”legend\_name”,linestyle=””,marker=””): legend name and other arguments are auxiliary
* plt.xlabel(“x label name”)
* plt.ylabel(“y label name”)
* plt.legend(): this shows the legend
* plt.text(x\_coordinate,y\_coordinate,“add text here”): we want to add text here
* plt.title(“title name”, font:…….) : … represents other arguments are also there
* plt.show() : write this after defining everthing otherwise the attributes will be added but not shown. It is like print().
* Plt.style.use(“style name”) : use style from other libraries , print(plt.style.available): show list of available styles

# Matplotlib(scatter, bar, stacked, histogram):

* plt.scatter(x\_values,y\_value,label=”label\_name”,color=”color\_name”,marker=”marker\_value”,alpha=value…): other methods are similar to the lineplot
* plt.scatter(x\_values,y\_value, alpha=value): alpha is transparency
* plt.bar(x\_values,y\_values): x is labeled so only add y label
* plt.bar(x,y,yerr=column\_name): yerr is the room of error column
* plt.bar(x,y,bottom=another\_column,label=”label\_name”….) : stacked bar chart but first you should define the bottom column before it as : plt.bar(x\_values,y\_values)
* plt.hist(column\_name, bins=value, range=(min,max)): by default bins\_value is 10 i.e. no of bars, the width of values will be created accordingly((max –min)/binsize). Range is zooming in to set the max and min value to see.
* If 2 colums data are to be viewed in the same histogram we have to add a **argument density= True** in both so that it displays both.

# Global, local and nonlocal

Global x  
nonlocal x : makes it accessible around that function in which it is enclosed.

x=10

def outer():

    x=5

    def inner():

        nonlocal x

        x=6

        print(x)

    inner()

    print(x)

outer()

print(x)

here, it will print 6 6 10. If we remove the nonlocal definition. The value of x inside the inner() will be 6 and will be local to that function only. It’s value in outer () will not be changed and will remain previous value as 5. And if printed it will return 6 5 10

# Passing value to inner function by calling outer function

# Define echo

def echo(n):

    """Return the inner\_echo function."""

    # Define inner\_echo

    def inner\_echo(word1):

        """Concatenate n copies of word1."""

        echo\_word = word1 \* n

        return echo\_word

    # Return inner\_echo

    return inner\_echo

# Call echo: twice

twice = echo(2)

# Call echo: thrice

thrice=echo(3)

# Call twice() and thrice() then print

print(twice('hello'), thrice('hello'))

# Flexible arguments(\*args,\*\*kwargs):

\*args allows multiple arguments(variable no of) to be passed without defining each one of them extensively.  
\*\*kwargs is used to pass a keyworded, variable-length argument list. It is used to pass values in (key,value) pairs as: **key=value**

def gibberish(\*args):

    """Concatenate strings in \*args together."""

    # Initialize an empty string: hodgepodge

    hodgepodge=str()

    # Concatenate the strings in args

    for word in args:

        hodgepodge += word

    # Return hodgepodge

    return hodgepodge

# Call gibberish() with one string: one\_word

one\_word = gibberish('luke')

# Call gibberish() with five strings: many\_words

many\_words = gibberish("luke", "leia", "han", "obi", "darth")

# Print one\_word and many\_words

print(one\_word)

print(many\_words)

Kwargs example:

def myFun(\*\*kwargs):

    for key, value in kwargs.items():

        print("%s == %s" % (key, value))

# Driver code

myFun(first='Geeks', mid='for', last='Geeks')

# Lambda function:

Allows to write function in a quick way. i.e. in a line  
we define by using ‘lambda’ keyword before the definition

max=lambda a,b:a if (a>b) else b

print(max(1,2))

# Create a list of strings: spells

spells = ["protego", "accio", "expecto patronum", "legilimens"]

# Use map() to apply a lambda function over spells: shout\_spells

shout\_spells = map(lambda item: item+'!!!', spells)

# Convert shout\_spells to a list: shout\_spells\_list

shout\_spells\_list=list(shout\_spells)

# Print the result

print(shout\_spells\_list)

# Create a list of strings: fellowship

fellowship = ['frodo', 'samwise', 'merry', 'pippin', 'aragorn', 'boromir', 'legolas', 'gimli', 'gandalf']

# Use filter() to apply a lambda function over fellowship: result

result = filter(lambda member:len(member)>6 , fellowship)

# Convert result to a list: result\_list

result\_list=list(result)

# Print result\_list

print(result\_list)