# complete python from scratch

**By Puru Sharma**

## variables

data/values can be stored in temporary storage spaces called variable

```
print("this is puru")
```

```
this is puru
```

```
student="puru"
student
```

```
'puru'
```

```
student="max"
student
```

```
'max'
```

## Data types in python

Every variable is associated with a data type

int =1233

float =3.14

boolean =True,False

string="puru"

In [13]:

```
p1=12
```

In [14]:

```
p1
```

Out[14]:

```
12
```

In [15]:

```
type(p1)
```

Out[15]:

```
int
```

In [16]:

```
p2=3.45
```

In [17]:

```
p2
```

Out[17]:

```
3.45
```

In [18]:

```
type(p2)
```

Out[18]:

```
float
```

In [19]:

```
p3="puru"
```

In [20]:

```
p3
```

Out[20]:

```
'puru'
```

In [21]:

```
type(p3)
```

Out[21]:

```
str
```

In [22]:

```
p4=True
```

In [23]:

```
p4
```

Out[23]:

```
True
```

In [24]:

```
type(p4)
```

Out[24]:

```
bool
```

In [25]:

```
a1=3+4j
```

In [26]:

```
a1
```

Out[26]:

```
(3+4j)
```

In [27]:

```
type(a1)
```

Out[27]:

```
complex
```

**operators in python**

**Arithmetic operator**

**Logical operator**

**Relation operator**

In [29]:

```
# arithmetic operator  (how to write comment in jupyter notebook)
```

In [30]:

```
a=1+10
b=45
```

In [31]:

```
a,b
```

Out[31]:

```
(11, 45)
```

In [32]:

```
a+b
```

Out[32]:

```
56
```

In [33]:

```
# relational operator
```

In [34]:

```
a=24
b=18
```

In [35]:

```
a>b
```

Out[35]:

```
True
```

In [36]:

```
a<b
```

Out[36]:

```
False
```

In [39]:

```
a!=b  # a is not equal to b  (!)
```

Out[39]:

```
True
```

In [40]:

```
# logical operators
```

In [41]:

```
a= True
b= False
```

In [42]:

```
a & a
```

Out[42]:

```
True
```

In [43]:

```
b& b
```

Out[43]:

```
False
```

In [44]:

```
a|b
```

Out[44]:

```
True
```

In [45]:

```
b|b
```

Out[45]:

```
False
```

### Python tokens

smallest meaningful component in a program

**Keywords**

**Identifiers**

**Literals**

**Operator**

### Python Keywords

```
Python Keywords

Python has a set of keywords that are reserved words that cannot be used as
variable names, function names, or any other identifiers

Keyword     Description
and       A logical operator
as        To create an alias
assert    For debugging
break    To break out of a loop
class    To define a class
continue  To continue to the next iteration of a loop
def      To define a function
del      To delete an object
elif     Used in conditional statements, same as else if
else     Used in conditional statements
```

| | |
|---|---|
| except | Used with exceptions, what to do when an exception occurs |
| False | Boolean value, result of comparison operations |
| finally | Used with exceptions, a block of code that will be executed no matter if there is an exception or not |
| for | To create a for loop |
| from | To import specific parts of a module |
| global | To declare a global variable |
| if | To make a conditional statement |
| import | To import a module |
| in | To check if a value is present in a list, tuple, etc. |
| is | To test if two variables are equal |
| lambda | To create an anonymous function |
| None | Represents a null value |
| nonlocal | To declare a non-local variable |
| not | A logical operator |
| or | A logical operator |
| pass | A null statement, a statement that will do nothing |
| raise | To raise an exception |
| return | To exit a function and return a value |
| True | Boolean value, result of comparison operations |
| try | To make a try...except statement |
| while | To create a while loop |
| with | Used to simplify exception handling |
| yield | To end a function, returns a generator |

**Python identifiers**

Identifier are names used for variables.functions or objects

Rules

no special character expect_(underscore) identifiers are case sensitive

first letter cannot be a digit

In [46]:

```
puru="rocks"    #case sensitive
```

In [51]:

```
Puru="result"   #case sensitive
```

In [52]:

```
Puru
```

Out[52]:

```
'result'
```

In [53]:

```
puru
```

Out[53]:

```
'rocks'
```

Python literals

literals these are constant in python

In [54]:

```
a=3.14
```

In [55]:

```
a
```

Out[55]:

```
3.14
```

In [56]:

```
type(a)
```

Out[56]:

```
float
```

**Python strings**

Strings are sequence of characters enclosed within single quotes(''),double quotes("") or triple quotes(''' ''')

In [57]:

```
str1='puru'
```

In [58]:

```
str1
```

Out[58]:

```
'puru'
```

In [59]:

```
str2="puru"
```

In [60]:

```
str2
```

Out[60]:

```
'puru'
```

In [61]:

```
str3="max"
```

In [62]:

```
str3
```

Out[62]:

```
'max'
```

**extracting individual character**

In [77]:

```
my_string="my name is puru sharma"
```

In [78]:

```
my_string[1]
```

Out[78]:

```
'y'
```

In [79]:

```
my_string[2]
```

Out[79]:

```
' '
```

In [80]:

```
my_string[-1]
```

Out[80]:

```
'a'
```

In [83]:

```
my_string[11:15]
```

Out[83]:

```
'puru'
```

**string functions**

finding length of string

In [85]:

```
len(my_string)
```

Out[85]:

```
22
```

converting string to lowercase

In [86]:

```
my_string.lower()
```

Out[86]:

```
'my name is puru sharma'
```

converting string to uppercase

In [87]:

```
my_string.upper()
```

Out[87]:

```
'MY NAME IS PURU SHARMA'
```

**replacing a substring**

In [90]:

```
my_string.replace('r','u')
```

Out[90]:

```
'my name is puru sharma'
```

**number of occurrences of substring**

In [91]:

```
new_string1="this is the great example of machine learning"
```

In [93]:

```
new_string1.count('e')
```

Out[93]:

```
6
```

### finding the index of substring

In [95]:

```
new_string1.find('s')
```

Out[95]:

```
3
```

### splitting a string

In [96]:

```
fruit='i like apples,mangoes,bananas'
fruit.split(',')
```

Out[96]:

```
['i like apples', 'mangoes', 'bananas']
```

In [97]:

```
student='puru,max,alice,alina,warner'
student.split(',')
```

Out[97]:

```
['puru', 'max', 'alice', 'alina', 'warner']
```

In [98]:

```
str5_final='president obama is the best president of us'
str5_final
```

Out[98]:

```
'president obama is the best president of us'
```

In [99]:

```
str5_final.split("s")
```

Out[99]:

```
['pre', 'ident obama i', ' the be', 't pre', 'ident of u', '']
```

### Data Structures in python

### tuple,set ,dictionary,list

### tuple is an ordered collection of elements enclosed within()

### tuples are immutable

### tup1=(1,'a',True) we can store heterogeneous data

In [100]:

```
tup1=(11,2,3.14,True,5+5j)
```

In [102]:

```
tup1
```

Out[102]:

```
(11, 2, 3.14, True, (5+5j))
```

In [103]:

```
type(tup1)
```

Out[103]:

```
tuple
```

### Tuple basic operations

### finding length of tuple

In [104]:

```
tup1=(1,"d",True,3)
```

In [107]:

```
len(tup1)
```

Out[107]:

```
4
```

### concatenating Tuples

In [108]:

```
tup1=(1,2,3)
tup2=(4,5,6)
tup1+tup2
```

Out[108]:

```
(1, 2, 3, 4, 5, 6)
```

### repeating Tuple Elements

In [109]:

```
tup1=('puru',450)
```

```
tup1*4
```

Out[109]:

```
('puru', 450, 'puru', 450, 'puru', 450, 'puru', 450)
```

**repeating and concatenating**

In [110]:

```
tup1
tup2
tup1*3+tup2
```

Out[110]:

```
('puru', 450, 'puru', 450, 'puru', 450, 4, 5, 6)
```

**Tuple Function**

In [112]:

```
#minimum value
tup1=(1,2,3,4,5)
min(tup1)
```

Out[112]:

```
1
```

In [113]:

```
#maximum value
tup1
max(tup1)
```

Out[113]:

```
5
```

**List in python**

**List is an ordered collection of elements enclosed within[]**

**List are mutable**

In [114]:

```
l1=[1,"puru",3.14,True]
l1
```

Out[114]:

```
[1, 'puru', 3.14, True]
```

In [115]:

```
type(l1)
```

Out[115]:

```
list
```

### extracting individual element

In [120]:

```
l1=[1,'w',2,'s','p']
l1[2]
```

Out[120]:

```
2
```

In [121]:

```
l1
l1[2:5]
```

Out[121]:

```
[2, 's', 'p']
```

### modifying a list

### changing the element at 0th index

In [122]:

```
l1=[1,"a",2,"b",3,"c"]
l1[0]=100
l1
```

Out[122]:

```
[100, 'a', 2, 'b', 3, 'c']
```

In [123]:

```
l1=[1,"a",2,"b",3,"c"]
l1[1]='b'
l1
```

Out[123]:

```
[1, 'b', 2, 'b', 3, 'c']
```

### popping the last element

### now we will see last element will remove with this method

In [124]:

```
l1=[1,"a",2,"b",3,"c","f"]
l1.pop()
l1
```

Out[124]:

```
[1, 'a', 2, 'b', 3, 'c']
```

### appending a new element

In [129]:

```
l2=[1,2,3,4,5,6,7,8,6]
```

In [130]:

```
l2.append("a")
```

In [131]:

```
l2
```

Out[131]:

```
[1, 2, 3, 4, 5, 6, 7, 8, 6, 'a']
```

### reversing element of a list

In [132]:

```
l1=[1,"a",2,"b",3,"c"]
l1.reverse()
l1
```

Out[132]:

```
['c', 3, 'b', 2, 'a', 1]
```

### sorting a list

In [133]:

```
l1=["nancy","admin","bravo"]
l1.sort()
l1
```

Out[133]:

```
['admin', 'bravo', 'nancy']
```

### inserting element at a specified index

In [135]:

```
l1=[1,"a",2,"b",3,"c"]
l1.insert(1,"puru")
l1
```

Out[135]:

```
[1, 'puru', 'a', 2, 'b', 3, 'c']
```

### Basic list operation

### concatenate lists

In [136]:

```
l1=[1,2,3]
l2=["a","b","c"]
l1+l2
```

Out[136]:

```
[1, 2, 3, 'a', 'b', 'c']
```

### repeating element

In [137]:

```
l1=[1,"a",True]
l1*3
```

Out[137]:

```
[1, 'a', True, 1, 'a', True, 1, 'a', True]
```

### Dictionary in python

### Dictionary is an unordered collection of key-value pairs

### enclosed with {}

### Dictionary is mutable

In [138]:

```
d1={'apple':35,'mango':40,'banana':40,'papaya':25}
print(d1)
```

```
{'apple': 35, 'mango': 40, 'banana': 40, 'papaya': 25}
```

In [139]:

```
type(d1)
```

Out[139]:

```
dict
```

### Extracting Keys and Values

In [140]:

```
deepanshi={"brother":1,"sister":0}
deepanshi.keys()
```

Out[140]:

```
dict_keys(['brother', 'sister'])
```

In [142]:

```
deepanshi.values()  #Extracting Values
```

Out[142]:

```
dict_values([1, 0])
```

### Modifying a Dictionary

### Adding a new element

In [143]:

```
fruit={"orange":30,"banana":20}
fruit["mango","papaya"]=50,30
fruit
```

Out[143]:

```
{'orange': 30, 'banana': 20, ('mango', 'papaya'): (50, 30)}
```

### Changing an existing element

In [144]:

```
fruit={"orange":30,"banana":20}
fruit["orange"]=100
fruit
```

Out[144]:

```
{'orange': 100, 'banana': 20}
```

### Dictionary Functions

### Update one dictionary's elements with another

In [145]:

```
fruit1={"orange":30,"banana":20}
fruit2={"apple":40,"papaya":25}
fruit1.update(fruit2)
fruit1
```

Out[145]:

```
{'orange': 30, 'banana': 20, 'apple': 40, 'papaya': 25}
```

### popping an element

In [146]:

```
fruit={"orange":30,"banana":20}
fruit.pop("orange")
fruit
```

Out[146]:

```
{'banana': 20}
```

### Set in python

### Set is an ordered and unindexed collection of elements enclosed with {}

### Duplicates are not allowed in Set

In [147]:

```
s1={1,3.14,"puru"}
```

In [148]:

```
s1
```

Out[148]:

```
{1, 3.14, 'puru'}
```

In [153]:

```
s2={1,1,1,3.14,"puru","puru"}  # duplicates are not allowed in sets
```

In [154]:

```
s2
```

Out[154]:

```
{1, 3.14, 'puru'}
```

### Set Operation

### Update one dictionary's elements with another

In [155]:

```
s1={1,"a",True,2,"b",False}
s1.add("hello")
s1
```

Out[155]:

```
{1, 2, False, 'a', 'b', 'hello'}
```

### Removing an element

In [156]:

```
s1={1,"a",True,2,"b",False}
s1.remove("b")
s1
```

Out[156]:

```
{1, 2, False, 'a'}
```

### updating multiple elements

In [157]:

```
s1={1,"a",True,2,"b",False}
s1.update([20,30])
s1
```

Out[157]:

```
{1, 2, 20, 30, False, 'a', 'b'}
```

### Set Function

### Union of two sets

In [158]:

```
s1={1,2,3}
s2={"a","b","c"}
s1.union(s2)
```

Out[158]:

```
{1, 2, 3, 'a', 'b', 'c'}
```

### intersection of two sets (common elements)

In [159]:

```
s1={1,2,34,5}
s2={4,5,6,7,8}
s1.intersection(s2)
```

Out[159]:

```
{5}
```

### If Statements

### Decision Making Statement

In [160]:

```
a=19
b=10
if a>b:
   print("b is greater than a")
else:
   print("b is not greater than a")
```

```
b is greater that a
```

In [161]:

```
a=10
b=20
c=30
if(a>b) &(a>c):
   print("a is the greatest")
elif(b>a)&(b>c):
   print("b is the greatest")
```

```
else:
   print("c is the greatest")
```

```
c is the greatest
```

### how to use if else with tuple

In [162]:

```
tup1=(1,'a','b')
```

In [163]:

```
if 'a' in tup1:
   print("value a is present in tup1")
else:
   print("value z is not present in tup1")
```

```
value a is present in tup1
```

In [164]:

```
if 'c' in tup1:
   print("value a is present in tup1")
else:
   print("value c is not present in tup1")
```

```
value c is not present in tup1
```

### if with list

In [175]:

```
l2=['a','b','c']
```

In [176]:

```
if l2[1]=='b':
   l2[1]='z'
```

In [177]:

```
l1
```

Out[177]:

```
['a', 'b', 'c']
```

### If with Dictionary

In [178]:

```
d1={'k1':49,'k2':35}
if d1['k2']==35:
   d1['k2']=d1['k2']+100
   print(d1)
```

```
{'k1': 49, 'k2': 135}
```

## Looping Statement

## Looping statements are used to repeat a task multiple times

In [1]:

```
i=1
while i<=10:
   print(i)
   i=i+1
```

```
1
2
3
4
5
6
7
8
9
10
```

In [2]:

```
i=1
n=2
while i<=10:
   print(n," * ",n*i)
   i=i+1
```

```
2  *  2
```

```
2  *  4
2  *  6
2  *  8
2  *  10
2  *  12
2  *  14
2  *  16
2  *  18
2  *  20
```

### While with list

In [3]:

```
l1=[1,2,3,4,5,6]
i=0
while i<len(l1):
   l1[i]=l1[i]+100
   i=i+1
```

In [4]:

```
print(l1)
```

```
[101, 102, 103, 104, 105, 106]
```

### Functions

Introduction to Functions What is a function in Python and how to create a function?

Functions will be one of our main building blocks when we construct larger and larger amounts of code to solve problems.

So what is a function?

A function groups a set of statements together to run the statements more than once. It allows us to specify parameters that can serve as inputs to the functions.

Functions allow us to reuse the code instead of writing the code again and again. If you recall strings and lists, remember that len() function is used to find the length of a string. Since checking the length of a sequence is a common task, you would want to write a function that can do this repeatedly at command.

Function is one of the most basic levels of reusing code in Python, and it will also allow us to start thinking of program design.

In [71]:

```
def test():
   print("this is my first function")
```

In [72]:

```
a=test()
```

```
this is my first function
```

In [73]:

```
type(a)
```

Out[73]:

```
NoneType
```

In [74]:

```
def test():
   return "this is my function"
```

In [75]:

```
a=test()
```

In [76]:

```
a
```

Out[76]:

```
'this is my function'
```

In [77]:

```
type(a)
```

Out[77]:

```
str
```

In [78]:

```
def test():
   return "this is my function"+  "puru"   # concatenations
```

In [79]:

```
a=test()
```

In [80]:

```
a
```

Out[80]:

```
'this is my functionpuru'
```

In [81]:

```
def test(x):
    return x*3
```

In [82]:

```
test() # you should always pass some kind of data then only will able to get some result
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-82-92c2989cca5d> in <module>
----> 1 test() # you should always pass some kind of data then only will able to get some result

TypeError: test() missing 1 required positional argument: 'x'
```

In [83]:

```
def test(x):
    return x*3
```

In [84]:

```
test(4)
```

Out[84]:

```
12
```

In [85]:

```
def test(y):
    return y*5,y+y
```

In [86]:

```
test("puru")
```

Out[86]:

```
('purupurupurupurupuru', 'purupuru')
```

In [87]:

```
type(test("puru"))
```

Out[87]:

```
tuple
```

In [88]:

```
def test(x):
        return x*5,x+x
```

In [89]:

```
a,b=test("puru")  # we can store data in individual variable
```

In [90]:

```
a
```

Out[90]:

```
'purupurupurupurupuru'
```

In [91]:

```
b
```

Out[91]:

```
'purupuru'
```

In [ ]:

In [92]:

```
def  test(x):
   for i in x:
       if i=="u":
           break
       print(i)
```

In [93]:

```
test("puru")
```

p

In [96]:

```
d={"key1":123,"key2":456}
```

In [98]:

```
d.items()
```

Out[98]:

```
dict_items([('key1', 123), ('key2', 456)])
```

In [2]:

```
def test(a):
    for i in a:
        print(i*1)
```

In [3]:

```
test([1,2,3,4])
```

```
1
2
3
4
```

In [4]:

```
type(test([1,2,3,4]))
```

```
1
2
3
4
```

Out[4]:

```
NoneType
```

```
def name_of_function(arg1,arg2):

''' this is where the function's document string (doc-string) goes '''

#do stuff here

#return desired result
```

In [6]:

```
def addition (a,b):
   return a+b
```

In [7]:

```
addition(6,9)
```

Out[7]:

```
15
```

In [8]:

```
def addition (a,b):
   return a+b
```

In [9]:

```
addition("puru","sharma")
```

Out[9]:

```
'purusharma'
```

### simple greeting function

In [13]:

```
def greeting(name):
   print('hello %d'%name)
```

In [14]:

```
greeting(7)  # if we pass the integer
```

```
hello 7
```

In [15]:

```
def greeting(name):
   print('hello %d'%name)
```

In [16]:

```
greeting("puru")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-e1b1bdebed9e> in <module>
----> 1 greeting("puru")
```

```
<ipython-input-15-87bdd55ccd5e> in greeting(name)
      1 def greeting(name):
----> 2     print('hello %d'%name)

TypeError: %d format: a number is required, not str
```

In [20]:

```
def greeting(name):   # if we pass the string
   print('hello %s'%name)
```

In [21]:

```
greeting("puru")
```

```
hello puru
```

## Using return

Let's see some examples that use a return statement. Return allows a function to "return" a result that can then be stored as a variable, or used in whatever manner a user wants.

## Addition function

In [22]:

```
def add_num(num1,num2):
   return num1+num2
```

In [23]:

```
add_num(9,8)
```

Out[23]:

```
17
```

In [2]:

```
a=input("enter the list of number").split(',')
a
```

```
enter the list of number 2,3,4,5,3,2
```

Out[2]:

```
['2', '3', '4', '5', '3', '2']
```

### how to make an calculator

In [8]:

```
def add_num(num1,num2):
   return num1+num2

def mul_num(num1,num2):
   return num1*num2
def sub_num(num1,num2):
   return num1-num2
def div_num(num1,num2):
   return num1/num2
```

In [7]:

```
div_num(2,5)
```

Out[7]:

```
0.4
```

In [9]:

```
sub_num(4,7)
```

Out[9]:

```
-3
```

In [20]:

```
def ageonmonths(age):
   c=age*12
   return c
```

In [22]:

```
e=ageonmonths(15)
print("the age in months",e)
```

```
the age in months 180
```

In [24]:

```
# how to check values
def check(a,b):
   return (a*b,a+b)
print(check(3,5),type(check(3,5)))
```

```
(15, 8) <class 'tuple'>
```

In [29]:

```
for i in  34: #int' object is not iterable


   print(i)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-29-d99de49863d3> in <module>
----> 1 for i in  34: #int' object is not iterable
      2 
      3     print(i)

TypeError: 'int' object is not iterable
```

In [30]:

```
next ("puru")  # str' object is not an iterator
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-30-fdd52f02660d> in <module>
----> 1 next ("puru")  # str' object is not an iterator

TypeError: 'str' object is not an iterator
```

In [31]:

```
range(10) # range function
```

Out[31]:

```
range(0, 10)
```

In [33]:

```
for i in range(6):
   print(i)
```

```
0
1
2
3
4
5
```

### Iterators and Generators

In this section, you will be learning the differences between iterations and generation in Python and also how to construct our own generators with the "yield" statement. Generators allow us to generate as we go along instead of storing everything in the memory.

We have learned how to create functions with "def" and the "return" statement. In Python, the Generator function allows us to write a function that can send back a value and then later resume to pick up where it was left. It also allows us to generate a sequence of values over time. The main difference in syntax will be the use of a **yield** statement.

In most aspects, a generator function will appear very similar to a normal function. The main difference is when a generator function is called and compiled they become an object that supports an iteration protocol. That means when they are called they don't actually return a value and then exit, the generator functions will automatically suspend and resume their execution and state around the last point of value generation.

The main advantage here is "state suspension" which means, instead of computing an entire series of values upfront, the generator functions can be suspended. To understand this concept better let's go ahead and learn how to create some generator functions.

In [34]:

```
# iterable object to iterator we can use function
```

In [35]:

```
a=iter(range(7))
```

In [39]:

```
next(a)  # print again and again
```

Out[39]:

```
3
```

In [40]:

```
next(a)
```

Out[40]:

```
4
```

In [41]:

```
next(a)
```

Out[41]:

```
5
```

In [2]:

```
# Generator function for the cube of numbers (power of 3)
def gencubes(n):
   for num in range(n):
       yield num**3
```

In [3]:

```
for x in gencubes(10):
   print(x)
```

```
0
1
8
27
64
125
216
343
512
```

```
729
```

In [3]:

```
#lambda function
g=lambda x:x*x*x
```

In [4]:

```
g(4)
```

Out[4]:

```
64
```

In [8]:

```
#lambda with filter
l1=[2,3,4,5,6]
list1=list(filter(lambda x:(x%2!=0),l1))
```

In [9]:

```
list1
```

Out[9]:

```
[3, 5]
```

In [17]:

```
#lambda with map
```

In [31]:

```
l1=[23,45,56,78,90]
list2=list(map(lambda x:x*2,l1))
```

In [32]:

```
list2
```

Out[32]:

```
[46, 90, 112, 156, 180]
```

In [33]:

```
from functools import reduce # consolidated result
```

In [34]:

```
l1=[1,2,3,4,5,6,7]
sum=(reduce(lambda x,y:x+y,l1))
```

In [35]:

```
sum
```

Out[35]:

28

## Object Oriented Programming and File I/O

**Object Oriented Programming (OOP)** is a programming paradigm that allows abstraction through the concept of interacting entities. This programming works contradictory to conventional models and is procedural, in which programs are organized as a sequence of commands or statements to perform.

We can think of an object as an entity that resides in memory, has a state and it's able to perform some actions.

More formally objects are entities that represent **instances** of a general abstract concept called **class**. In Python, "attributes" are the variables defining an object state and the possible actions are called "methods".

In Python, everything is an object, also classes and functions.

### what is class

class is template/blueprint for real-world entities,class is user defined type

Properties

color

cost

battery life
Behavior

make calls watch videos play games

### objects are specific instance of a class

phone ----->apple motorola mi samsung

### creating the first class

In [63]:

```
class Phone:   # creating first class
   def make_call(self):  # if we want to invoke inbuilt parameter through
```

```
object where we use self.when we write first  method in class where first parameter should be self
        print("make an phone call")
    def play_game(self):
        print("playing game")
    def play_song(self):
        print("playing song")
```

In [69]:

```
# instantiating the p1 object
p1=Phone()
```

In [70]:

```
# invoking methods through object
p1.make_call()
```

```
make an phone call
```

In [71]:

```
p1.play_game()
```

```
playing game
```

In [72]:

```
p1.play_song()
```

```
playing song
```

### Adding parameter to the class

In [77]:

```
class Phone:   #setting and returning the attribute values
    def set_color(self,color):
        self.color=color
    def set_cost(self,cost):
        self.cost=cost
    def show_color(self):
        return self.color
    def show_cost(self):
        return self.cost



    def make_call(self):
        print("making phone call")
    def play_game(self):
        print("playing game")
```

In [85]:

```
p2=Phone()
```

In [86]:

```
p2.set_color("red")
```

In [87]:

```
p2.set_cost(34000)
```

In [88]:

```
p2.show_color()
```

Out[88]:

```
'red'
```

In [89]:

```
p2.show_cost()
```

Out[89]:

```
34000
```

In [90]:

```
p2.play_game()
```

```
playing game
```

**creating a class with constructor**

In [102]:

```
class Employee:
    def __init__(self,name,age,salary,gender):  # init method act as the constructor



        self.name=name
        self.age=age
        self.salary=salary
        self.gender=gender

    def employee_details(self):
        print("Name of employee is",self.name)
        print("Age of employee is",self.age)
        print("Salary of employee is",self.salary)
        print("Gender of employee is",self.gender)
```

In [103]:

```
# instantiating the e1 object
e1=Employee("puru",23,250000,"male")
```

In [105]:

```
e1.employee_details() # invoking the employee_details method
```

```
Name of employee is puru
Age of employee is 23
Salary of employee is 250000
Gender of employee is male
```

### Inheritance in python

with inheritance one class can derive the properties of another class

In [112]:

```
# inheritance example

class Vehicle:   # creating the base class
    def __init__(self,milage,cost):
        self.milage=milage
        self.cost=cost



    def show_details(self):
        print("i am a Vehicle")
        print("Milage of vehicle is",self.milage)
        print("cost of vehicle is",self.cost)
```

In [114]:

```
v1=Vehicle(500,400)
```

In [117]:

```
v1.show_details()  # instantiating the object for base class
```

```
i am a Vehicle
Milage of vehicle is 500
cost of vehicle is 400
```

In [119]:

```
class Car(Vehicle):  # creating the child class
    def show_car(self):
        print("i am a car")
```

In [121]:

```
c1=Car(288,1234)
```

In [123]:

```
c1.show_details() # instantiating the object for child class
```

```
i am a Vehicle
Milage of vehicle is 288
cost of vehicle is 1234
```

In [126]:

```
c1.show_car()  # invoking the child class method
```

```
i am a car
```

**Overriding init method**

In [128]:

```
class Car(Vehicle): # overriding init method
    def __init__(self,milage,cost,tyres,hp):
        super().__init__(milage,cost)
        self.tyres=tyres
        self.hp=hp
    def show_car_details(self):
        print("i am a car")
        print("number of tyres are",self.tyres)
        print("value of horse power is",self.hp)
```

In [129]:

```
# invoking show_details() method from parent class
```

In [130]:

```
c1=Car(20,13456,5,600)
```

In [131]:

```
c1.show_details()
```

```
i am a Vehicle
Milage of vehicle is 20
cost of vehicle is 13456
```

In [132]:

```
c1.show_car_details()
```

```
i am a car
number of tyres are 5
value of horsepower is 600
```

## *args and **kwargs in Python

The special syntax _*args_ in function definitions in python is used to pass a variable number of arguments to a function. It is used to pass a non-keyworded, variable-length argument list.

- The syntax is to use the symbol * to take in a variable number of arguments; by convention, it is often used with the word args.
- What _*args_ allows you to do is take in more arguments than the number of formal arguments that you previously defined. With

**args*, any number of extra arguments can be tacked on to your current formal parameters (including zero extra arguments).

- For example : we want to make a multiply function that takes any number of arguments and able to multiply them all together. It can be done using *args.
- Using the *, the variable that we associate with the * becomes an iterable meaning you can do things like iterate over it, run some higher order functions such as map and filter, etc.

**Example for usage of *arg:**

```
 # Python program to illustrate
# *args for variable number of arguments
def myFun(*argv):
        for arg in argv:
                print (arg)

myFun('Hello', 'Welcome', 'to', 'Puru Sharma')
```

**Output:**

```
Hello
Welcome
To
Puru Sharma
```

```
# Python program to illustrate
# *args with first extra argument
def myFun(arg1, *argv):
        print ("First argument :", arg1)
        for arg in argv:
                print("Next argument through *argv :", arg)
```

```
myFun('Hello', 'Welcome', 'to', 'Puru Sharma')
```

```
First argument : Hello
Next argument through *argv : Welcome
Next argument through *argv : to
Next argument through *argv : Puru Sharma
```

## **kwargs

The special syntax ***kwargs* in function definitions in python is used to pass a keyworded, variable-length argument list. We use the name *kwargs* with the double star. The reason is because the double star allows us to pass through keyword arguments (and any number of them).

- A keyword argument is where you provide a name to the variable as you pass it into the function.
- One can think of the *kwargs* as being a dictionary that maps each keyword to the value that we pass alongside it. That is why when we iterate over the *kwargs* there doesn't seem to be any order in which they were printed out.

```
# Python program to illustrate

# *kargs for variable number of keyword arguments


def myFun(**kwargs):

	for key, value in kwargs.items():
        print ("%s == %s" %(key, value))


# Driver code
myFun(first ='deepak', mid ='singh', last='tomar')
```

**Output:**

```
last == deepak
mid == singh
first == tomar
```

**multiple inheritance**

in multiple inheritance,to child inherits from more than 1 parent class

| Parent 1 | |Parent 2 |

child

In [133]:

```
# multiple inheritance python

# parent class one

class Parent1():
    def assign_string_one(self,str1):
        self.str1=str1
    def show_string_one(self):
        return self.str1
```

In [134]:

```
# parent class two
class Parent2():
    def assign_string_two(self,str2):
        self.str2=str2
    def show_string_two(self):
        return self.str2
```

In [140]:

```
#child class
class Derived(Parent1,Parent2):
    def assign_string_three(self,str3):
        self.str3=str3
    def show_string_three(self):
        return self.str3
```

In [141]:

```
#instantiating object of child class
d1=Derived()
```

In [142]:

```
d1.assign_string_one("one")
d1.assign_string_two("two")
d1.assign_string_three("three")
```

In [143]:

```
# invoking methods
d1.show_string_one()
```

Out[143]:

```
'one'
```

In [144]:

```
d1.show_string_two()
```

Out[144]:

```
'two'
```

In [145]:

```
d1.show_string_three()
```

Out[145]:

```
'three'
```

### Multilevel inheritance

in multilevel inheritance, we have parent, child,grand-child relationship

parent- child- grand-child

In [169]:

```
#parent class
class Parent():
   def assign_name(self,name):
       self.name=name
   def show_name(self):
       return self.name
```

In [170]:

```
#child class
```

```
class Child(Parent):
   def assign_age(self,age):
       self.age=age

   def show_age(self):
       return self.age
```

In [186]:

```
# Grand child class
class GrandChild(Child):
   def assign_gender(self,gender):
       self.gender=gender

   def show_gender(self):
       return self.gender
```

In [187]:

```
gc=GrandChild()
```

In [188]:

```
gc.assign_name("puru")
```

In [189]:

```
gc.assign_age(23)
```

In [190]:

```
gc.assign_gender("male")
```

In [191]:

```
gc.show_name()
```

Out[191]:

```
'puru'
```

In [192]:

```
gc.show_age()
```

Out[192]:

```
23
```

In [193]:

```
gc.show_gender()
```

Out[193]:

```
'Male'
```

**Protect your abstraction**

Here the instance attributes shouldn't be accessible by the end user of an object as they are powerful means of abstraction they should not reveal the internal implementation detail. In Python, there is no specific strict mechanism to protect object attributes but the official guidelines suggest that a variable that has an underscore prefix should be treated as 'Private'.

Moreover prepending two underscores to a variable name makes the interpreter mangle a little the variable name.

**Example 1**

```
class Person:
    def __init__(self, name, surname, year_of_birth):
        self._name = name
        self._surname = surname
        self._year_of_birth = year_of_birth

    def age(self, current_year):
        return current_year - self._year_of_birth

    def __str__(self):
        return "%s %s and was born %d." \
                % (self._name, self._surname, self._year_of_birth)
```

```
alec = Person("Alec", "Baldwin", 1958)
print(alec)
print(alec._name)
```

**Output**

```
Alec Baldwin and was born 1958.
Alec
```

**Example 2**

```
class Person:
    def __init__(a, name, surname, year_of_birth):
        a.__name = name
        self.__surname = surname
        self.__year_of_birth = year_of_birth

    def age(self, current_year):
        return current_year - self.__year_of_birth

    def __str__(self):
        return "%s %s and was born %d." \
                % (self.__name, self.__surname, self.__year_of_birth)

alec = Person("Alec", "Baldwin", 1958)
```

```
print(alec._Person__name)
```

_dict__ is a special attribute is a dictionary containing each attribute of an object. We can see that prepending two underscores every key has _ClassName__ prepended.

### Encapsulation

Encapsulation is another powerful way to extend a class which consists on wrapping an object with a second one. There are two main reasons to use encapsulation:

- Composition
- Dynamic Extension

### Composition

The abstraction process relies on creating a simplified model that remove useless details from a concept. In order to be simplified, a model should be described in terms of other simpler concepts. For example, we can say that a car is composed by:

- Tyres
- Engine
- Body

And break down each one of these elements in simpler parts until we reach primitive data.

### Let's take an example

```
class Tyres:

    def __init__(self, branch, belted_bias, opt_pressure):
        self.branch = branch
        self.belted_bias = belted_bias
        self.opt_pressure = opt_pressure

    def __str__(self):
        return ("Tyres: \n \tBranch: " + self.branch +
               "\n \tBelted-bias: " + str(self.belted_bias) +
               "\n \tOptimal pressure: " + str(self.opt_pressure))

class Engine:
    def __init__(self, fuel_type, noise_level):
        self.fuel_type = fuel_type
        self.noise_level = noise_level

    def __str__(self):
        return ("Engine: \n \tFuel type: " + self.fuel_type +
                "\n \tNoise level:" + str(self.noise_level))

class Body:
    def __init__(self, size):
        self.size = size
    def __str__(self):
        return "Body:\n \tSize: " + self.size

class Car:
    def __init__(self, tyres, engine, body):
        self.tyres = tyres
        self.engine = engine
        self.body = body

    def __str__(self):
        return str(self.tyres) + "\n" + str(self.engine) + "\n" + str(self.body)


t = Tyres('Pirelli', True, 2.0)
e = Engine('Diesel', 3)
b = Body('Medium')
c = Car(t, e, b)
print(c)
```

**Output**

```
Tyres:
        Branch: Pirelli
        Belted-bias: True
        Optimal pressure: 2.0
Engine:
        Fuel type: Diesel
        Noise level:3
Body:
        Size: Medium
```

**Dynamic Extension**

Sometimes it's necessary to model a concept that may be a subclass of another one, but it isn't possible to know which class should be its superclass until runtime.

**Example**

Suppose we want to model a simple dog school that trains instructors too. It will be nice to re-use Person and Student but students can be dogs or peoples. So we can remodel it this way:

```
class Dog:
    def __init__(self, name, year_of_birth, breed):
        self._name = name
        self._year_of_birth = year_of_birth
        self._breed = breed

    def __str__(self):
        return "%s is a %s born in %d." % (self._name, self._breed, self._year_of_birth)

kudrjavka = Dog("Kudrjavka", 1954, "Laika")
print(kudrjavka)
```

**Output**

```
Kudrjavka is a Laika born in 1954.
```

**Example 2**

```
class Student:
    def __init__(self, anagraphic, student_id):
        self._anagraphic = anagraphic
        self._student_id = student_id
    def __str__(self):
        return str(self._anagraphic) + " Student ID: %d" % self._student_id


alec_student = Student("dsfs",1)
kudrjavka_student = Student(kudrjavka, 2)

print(alec_student)
print(kudrjavka_student)
```

**Output**

```
dsfs Student ID: 1
Kudrjavka is a Laika born in 1954. Student ID: 2
```

**Polymorphism and DuckTyping**

Python uses dynamic typing which is also called duck typing. If an object implements a method you can use it, irrespective of the type. This is different from statically typed languages, where the type of a construct need to be explicitly declared. Polymorphism is the ability to use the same syntax for objects of different types:

```
def summer(a, b):
    return a + b

print(summer(1, 1))
print(summer(["a", "b", "c"], ["d", "e"]))
print(summer("abra", "cadabra"))
```

**Output**

```
2
['a', 'b', 'c', 'd', 'e']
Abracadabra
```

### How long does a class should be?

There is an Object Oriented Programming (OOP) principle called Single Responsibility Principle (SRP) and it states: "A class should have one single responsibility" or "A class should have only one reason to change".

If you come across a class which doesn't follow the SRP principle, you should spilt it. You will be grateful to SRP during your software maintenance.

### Files

Python uses file objects to interact with the external files on your computer. These file objects cab be of any file format on your computer i.e. can be an audio file, a text file, emails, Excel documents, etc. Note that You will probably need to install certain libraries or modules to interact with those various file types, but they are easily available. (We will cover downloading modules later on in the course).

Python has a built-in open function that allows us to open and play with basic file types. First we will need a file though. We're going to use some iPython magic to create a text file!

### iPython Writing a File

### iPython Writing a File

In [58]:

```
%%writefile test.txt
Hello, this is a quick test file hjgtyudfyffhgghghhfch
```

```
Overwriting test.txt
```

In [55]:

```
pwd()
```

Out[55]:

```
'/Users/sudhanshukumar/Downloads/acad material/ACD_MDS_Offline_V2_Session_2_Code (5)'
```

### Python Opening a file

We can open a file with the open() function. This function also takes in arguments (also called parameters). Let's see how this is used:

In [65]:

```
# Open the text.txt we made earlier
my_file = open('test.txt')
```

In [66]:

```
# We can now read the file
my_file.read()
```

Out[66]:

```
'Hello, this is a quick test file hjgtyudfyffhgghghhfch\n'
```

In [64]:

```
# But what happens if we try to read it again?
my_file.read()
```

Out[64]:

''

This happens because you can imagine the reading "cursor" is at the end of the file after having read it. So there is nothing left to read. We can reset the "cursor" like this:

In [72]:

```
# Seek to the start of file (index 0)
my_file.seek(20)
```

Out[72]:

```
20
```

In [73]:

```
# Now read again
my_file.read()
```

Out[73]:

```
'ck test file hjgtyudfyffhgghghhfch\n'
```

In order to not have to reset every time, we can also use the readlines method. Use caution with large files, since everything will be held in memory. We will learn how to iterate over large files later in the course.

In [40]:

```
# Seek to the start of file (index 0)
my_file.seek(0)
```

Out[40]:

```
0
```

In [41]:

```
# Readlines returns a list of the lines in the file.
my_file.readlines()
```

Out[41]:

```
['Hello, this is a quick test file']
```

### Writing to a File

By default, using the open() function will only allow us to read the file, we need to pass the argument 'w' to write over the file. For example:

In [74]:

```
# Add the second argument to the function, 'w' which stands for write
```

```
my_file = open('test.txt','w+')
```

In [75]:

```
# Write to the file
my_file.write('This is a new line')
```

Out[75]:

```
18
```

In [76]:

```
# Seek to the start of file (index 0)
my_file.seek(0)
```

Out[76]:

```
0
```

In [77]:

```
# Read the file
my_file.read()
```

Out[77]:

```
'This is a new line'
```

### Iterating through a File

Let's get a quick preview of a for loop by iterating over a text file. First, let's make a new text file with some iPython Magic:

In [78]:

```
%%writefile test.txt
First Line
Second Line
```

```
Overwriting test.txt
```

In [79]:

```
my_file = open('test.txt')
my_file.read()
```

Out[79]:

```
'First Line\nSecond Line\n'
```

Now we can use a little bit of flow to tell the program to for through every line of the file and do something:

In [80]:

```
for line in open('test.txt'):
    print(line)
```

```
First Line


Second Line
```

In [50]:

```
# Pertaining to the first point above
for asdf in open('test.txt'):
    print(asdf)
```

```
First Line


Second Line
```

### StringIO

The StringIO module implements an in-memory filelike object. This object can then be used as input or output to most functions that would expect a standard file object.

The best way to show this is by example:

In [83]:

```
from io import StringIO
```

In [84]:

```
# Arbitrary String
message = 'This is just a normal string.'
```

In [85]:

```
# Use StringIO method to set as file object
```

```
f = StringIO(message)
```

Now we have an object *f* that we will be able to treat just like a file. For example:

In [86]:

```
f.read()
```

Out[86]:

```
'This is just a normal string.'
```

We can also write to it

In [87]:

```
f.write(' Second line written to file like object')
```

Out[87]:

```
40
```

In [88]:

```
# Reset cursor just like you would a file
f.seek(5)
```

Out[88]:

```
5
```

In [89]:

```
# Read again
f.read()
```

Out[89]:

```
'is just a normal string. Second line written to file like object'
```

In [ ]:

**In [ ]:**

**Libraries in python**

Python library is a collection of functions and methods that allows you to perform any actions without writing your code

**Numpy,Matplotlib,Pandas**

In [198]:

```
pip install matplotlib  # downloaded package
```

```
Collecting matplotlib
  Cache entry deserialization failed, entry ignored
  Cache entry deserialization failed, entry ignored
  Downloading
https://files.pythonhosted.org/packages/93/4b/52da6b1523d5139d04e02d9e26ceda6146b48f2a4e5d2abfdf1c7bac8c40/matplotlib-3.2.1-cp36-cp36m-manylinux1_x86_64.whl (12.4MB)
    100% |████████████████████████████████| 12.4MB
81kB/s eta 0:00:01   41% |█████████████▏                  | 5.2MB
2.7MB/s eta 0:00:03    49% |███████████████▉                | 6.1MB
4.0MB/s eta 0:00:02    57% |██████████████████▍             | 7.1MB
3.7MB/s eta 0:00:02
Collecting kiwisolver>=1.0.1 (from matplotlib)
  Cache entry deserialization failed, entry ignored
  Cache entry deserialization failed, entry ignored
  Downloading
https://files.pythonhosted.org/packages/ae/23/147de658aabbf968324551ea22c0c13a00284c4ef49a77002e91f79657b7/kiwisolver-1.2.0-cp36-cp36m-manylinux1_x86_64.whl (88kB)
    100% |████████████████████████████████| 92kB
```

```
2.6MB/s ta 0:00:011
Collecting numpy>=1.11 (from matplotlib)
  Cache entry deserialization failed, entry ignored
  Downloading
https://files.pythonhosted.org/packages/b3/a9/b1bc4c935ed063766bce7d3
e8c7b20bd52e515ff1c732b02caacf7918e5a/numpy-1.18.5-cp36-cp36m-
manylinux1_x86_64.whl (20.1MB)
    100% |████████████████████████████████| 20.1MB
48kB/s eta 0:00:011   65% |████████████████████▉           |
13.1MB 3.9MB/s eta 0:00:02
Collecting cycler>=0.10 (from matplotlib)
  Cache entry deserialization failed, entry ignored
  Cache entry deserialization failed, entry ignored
  Downloading
https://files.pythonhosted.org/packages/f7/d2/e07d3ebb2bd7af696440ce7e
754c59dd546ffe1bbe732c8ab68b9c834e61/cycler-0.10.0-py2.py3-none-
any.whl
Collecting pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 (from matplotlib)
  Cache entry deserialization failed, entry ignored
  Cache entry deserialization failed, entry ignored
  Downloading
https://files.pythonhosted.org/packages/8a/bb/488841f56197b13700afd565
8fc279a2025a39e22449b7cf29864669b15d/pyparsing-2.4.7-py2.py3-none-
any.whl (67kB)
    100% |████████████████████████████████| 71kB
3.4MB/s ta 0:00:01
Collecting python-dateutil>=2.1 (from matplotlib)
  Cache entry deserialization failed, entry ignored
  Using cached
https://files.pythonhosted.org/packages/d4/70/d60450c3dd48ef8758692420
7ae8907090de0b306af2bce5d134d78615cb/python_dateutil-2.8.1-py2.py3-
none-any.whl
Collecting six (from cycler>=0.10->matplotlib)
  Cache entry deserialization failed, entry ignored
  Using cached
```

```
https://files.pythonhosted.org/packages/ee/ff/48bde5c0f013094d729fe4b03
16ba2a24774b3ff1c52d924a8a4cb04078a/six-1.15.0-py2.py3-none-
any.whl
Installing collected packages: kiwisolver, numpy, six, cycler, pyparsing,
python-dateutil, matplotlib
Successfully installed cycler-0.10.0 kiwisolver-1.2.0 matplotlib-3.2.1 numpy-
1.18.5 pyparsing-2.4.7 python-dateutil-2.8.1 six-1.15.0
Note: you may need to restart the kernel to use updated packages.
```

In [ ]:

```
import matplotlib
```

### Python Numpy

Numpy stands for numerical python and is the core library for numeric and scientific computing.

it consists of a multidimensional array object and a collection of routines for processing those arrays.

### how to create numpy array

In [14]:

```
#single-dimensional array

import numpy as np  # numpy as a np means alias
n1=np.array([10,20,30,34])
n1
```

Out[14]:

```
array([10, 20, 30, 34])
```

In [15]:

```
# Multi dimensional array
import numpy as np
n2=np.array([[10,20,30,40],[98,87,76,43]])
n2
```

Out[15]:

```
array([[10, 20, 30, 40],
       [98, 87, 76, 43]])
```

## Initializing Numpy Array

### initializing numpy array with zeros

In [16]:

```
import numpy as np
```

In [17]:

```
n1=np.zeros((1,3))
n1
```

Out[17]:

```
array([[0., 0., 0.]])
```

In [18]:

```
import numpy as np
n1=np.zeros((5,5))
n1
```

Out[18]:

```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

In [19]:

```
import numpy as np
n1=np.zeros((10,10))
n1
```

Out[19]:

```
array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

**initializing Numpy array with same number**

In [21]:

```
import numpy as np
n1=np.full((2,2),10)
n1
```

Out[21]:

```
array([[10, 10],
       [10, 10]])
```

**initializing Numpy array within a range**

In [23]:

```
import numpy as np
n1=np.arange(10,20)
n1
```

Out[23]:

```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

In [25]:

```
import numpy as np
n1=np.arange(10,40,5)
n1
```

Out[25]:

```
array([10, 15, 20, 25, 30, 35])
```

In [28]:

```
import numpy as np
n1=np.arange(10,245)
n1
```

Out[28]:

```
array([ 10,  11,  12,  13,  14,  15,  16,  17,  18,  19,  20,  21,  22,
        23,  24,  25,  26,  27,  28,  29,  30,  31,  32,  33,  34,  35,
        36,  37,  38,  39,  40,  41,  42,  43,  44,  45,  46,  47,  48,
        49,  50,  51,  52,  53,  54,  55,  56,  57,  58,  59,  60,  61,
        62,  63,  64,  65,  66,  67,  68,  69,  70,  71,  72,  73,  74,
        75,  76,  77,  78,  79,  80,  81,  82,  83,  84,  85,  86,  87,
        88,  89,  90,  91,  92,  93,  94,  95,  96,  97,  98,  99, 100,
       101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113,
       114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126,
       127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139,
       140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152,
       153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165,
       166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178,
       179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191,
       192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204,
       205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217,
       218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230,
       231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243,
       244])
```

### initializing Numpy array with random numbers

In [31]:

```
import numpy as np
n1=np.random.randint(1,100,20)
n1
```

Out[31]:

```
array([84, 60, 50, 56, 26, 78, 39, 40, 82, 88,  1, 70, 18, 47, 99,  9, 80,
       58, 91, 90])
```

### checking the shape of Numpy arrays

In [37]:

```
import numpy as np
n1=np.array([[1,2,3],[4,5,6]])
n1.shape
```

Out[37]:

```
(2, 3)
```

In [38]:

```
n1
```

Out[38]:

```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [35]:

```
n1.shape=(3,2)
n1.shape
```

Out[35]:

```
(3, 2)
```

In [36]:

```
n1
```

Out[36]:

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

**joining numpy arrays**

In [42]:

```
#vstack()    veritical
import numpy as np
n1=np.array([1,2,3,4])
n2=np.array([4,5,6,7])

np.vstack((n1,n2))
```

Out[42]:

```
array([[1, 2, 3, 4],
       [4, 5, 6, 7]])
```

In [44]:

```
#hstack() horizontal
import numpy as np
n1=np.array([10,20,30,40])
n2=np.array([50,60,70,80])

np.hstack((n1,n2))
```

Out[44]:

```
array([10, 20, 30, 40, 50, 60, 70, 80])
```

In [46]:

```
#column_stack() column

import numpy as np
n1=np.array([23,34,56,67])
n2=np.array([12,343,56,87])
np.column_stack((n1,n2))
```

Out[46]:

```
array([[ 23,  12],
       [ 34, 343],
       [ 56,  56],
       [ 67,  87]])
```

### Numpy Intersection & Difference

In [56]:

```
import numpy as np  # common
n1=np.array([12,34,62,234,22])
n2=np.array([45,65,62,44,40])
```

In [57]:

```
np.intersect1d(n1,n2)
```

Out[57]:

```
array([62])
```

In [58]:

```
import numpy as np  # common
n1=np.array([12,45,62,234,22])
n2=np.array([45,65,62,44,40])
```

In [59]:

```
np.setdiff1d(n1,n2)
```

Out[59]:

```
array([ 12,  22, 234])
```

In [60]:

```
import numpy as np  # common
n1=np.array([12,34,62,234,22,400])
n2=np.array([45,65,62,44,40,400])
```

In [62]:

```
np.setdiff1d(n2,n1)
```

Out[62]:

```
array([40, 44, 45, 65])
```

**addition of numpy arrays**

In [64]:

```
import numpy as np
n1=np.array([10,20])
n2=np.array([40,30])
np.sum([n1,n2])
```

Out[64]:

```
100
```

In [65]:

```
np.sum([n1,n2],axis=0)
```

Out[65]:

```
array([50, 50])
```

In [66]:

```
np.sum([n1,n2],axis=1)
```

Out[66]:

```
array([30, 70])
```

**Numpy array mathematics**

In [70]:

```
#basic addition

import numpy as np
n1=np.array([10,20,30])
n1=n1+45
n1
```

Out[70]:

```
array([55, 65, 75])
```

In [72]:

```
#basic subtraction
import numpy as np
```

```
n1=np.array([34,56,67])
n1=n1-1
n1
```

Out[72]:

```
array([33, 55, 66])
```

In [73]:

```
#basic multiplication
import numpy as np
n1=np.array([43,45,34])
n1=n1*2
```

In [74]:

```
n1
```

Out[74]:

```
array([86, 90, 68])
```

In [76]:

```
#basic division
import numpy as np
n1=np.array([23,45,67,23,567,78])
n1=n1/4
```

In [77]:

```
n1
```

Out[77]:

```
array([  5.75,  11.25,  16.75,   5.75, 141.75,  19.5 ])
```

### Numpy math functions

### mean

In [79]:

```
import numpy as np
n1=np.array([10,20,30,40])
np.mean(n1)
```

Out[79]:

```
25.0
```

### standard deviation

In [83]:

```
import numpy as np
n=np.array([2,3,4,5,6,7,9,5,3,23])
np.std(n)
```

Out[83]:

```
5.780138406647371
```

**median**

In [84]:

```
import numpy as np
n2=np.array([4,33,2,5,6])
np.median(n2)
```

Out[84]:

```
5.0
```

**Numpy Save and load**

In [85]:

```
import numpy as np
n1=np.array([3,4,5,6,7,8])
np.save('my_numpy',n1)
```

In [86]:

```
n1
```

Out[86]:

```
array([3, 4, 5, 6, 7, 8])
```

In [89]:

```
n2=np.load('my_numpy.npy')  # npy extension
n2
```

Out[89]:

```
array([3, 4, 5, 6, 7, 8])
```

**Python Pandas**

pandas stands for panel data and is te core library for data manipulation and data analysis.

it consist of single and multi-dimensional data-structure for datamanipulation.

```
Single dimensional          Multidimensional
     |                              |
     |
Series object                    Data frame
```

**series object is one dimensional labeled array**

In [96]:

```
import pandas as pd
s1=pd.Series([1,2,3,4,5,6,7,5,34])
s1
```

Out[96]:

```
0     1
1     2
2     3
3     4
4     5
5     6
6     7
7     5
8    34
dtype: int64
```

In [95]:

```
type(s1)
```

Out[95]:

```
pandas.core.series.Series
```

**Changing Index**

In [102]:

```
import pandas as pd
s1=pd.Series([3,4,5,6,7],index=['a','b','c','d','e'])
s1
```

Out[102]:

```
a    3
b    4
```

```
c    5
d    6
e    7
dtype: int64
```

### Series object from Dictionary

you can also create a series object from a dictionary!!

In [104]:

```
import pandas as pd
pd.Series({'a':10,'b':34,'c':23})
```

Out[104]:

```
a    10
b    34
c    23
dtype: int64
```

In [107]:

```
import pandas as pd # you can change the index positions
pd.Series({'a':10,'b':34,'c':23},index=['b','c','d','a'])
```

Out[107]:

```
b    34.0
c    23.0
d     NaN
a    10.0
dtype: float64
```

### Extracting individual elements

In [108]:

```
#extracting a single element
```

In [111]:

```
s1=pd.Series([1,2,3,4,5,6,7,8,9])
s1[4]
```

Out[111]:

```
5
```

In [112]:

```
#extracting a sequence of elements
```

In [113]:

```
s1=pd.Series([1,2,3,4,5,6,7,8,9])
s1[:5]
```

Out[113]:

```
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

In [114]:

```
#extracting elements from back
```

In [115]:

```
s1=pd.Series([1,2,3,4,5,6,7,8,9])
s1[-3:]
```

Out[115]:

```
6    7
7    8
8    9
dtype: int64
```

**Basic math operations on series**

In [116]:

```
#adding a scalar value to series elements
```

In [117]:

```
s1+5
```

Out[117]:

```
0     6
1     7
2     8
3     9
4    10
5    11
```

```
6    12
7    13
8    14
dtype: int64
```

In [118]:

```
#adding two series objects
```

In [120]:

```
s1=pd.Series([1,2,3,4,5,6,7,8,9])
s2=pd.Series([10,20,30,40,50,60])
```

In [121]:

```
s1+s2
```

Out[121]:

```
0    11.0
1    22.0
2    33.0
3    44.0
4    55.0
5    66.0
6     NaN
7     NaN
8     NaN
dtype: float64
```

In [122]:

```
s1*s2
```

Out[122]:

```
0     10.0
1     40.0
2     90.0
3    160.0
4    250.0
5    360.0
6      NaN
7      NaN
8      NaN
```

```
dtype: float64
```

## Pandas Dataframe

## Dataframe is 2-dimensional labelled data-structure

## A data-frame comprises of rows and columns

In [123]:

```
# this is how you can create a data frame
```

In [131]:

```
import pandas as pd
pd.DataFrame({'Name':['puru','shivi','viyan','rishi','max','alina'],"marks":[23,45,67
,34,56,78]})
```

Out[131]:

|   | Name | marks |
|---|---|---|
| **0** | puru | 23 |
| **1** | shivi | 45 |
| **2** | viyan | 67 |
| **3** | rishi | 34 |
| **4** | max | 56 |
| **5** | alina | 78 |

Type *Markdown* and LaTeX:
α
2