Printing a Message

For printing a message in python we use a builtin function called print() function in which we pass the string in the arguments during calling of the print() function in your python program. Just take an example of the string: "Hello Bhaiyo!". As we know that, string is enclosed with double quotation mark.

In [4]:
```
print("Hello Bhaiyo")
```

```
Hello Bhaiyo
```

Variables

There is a concept of variable comes in programming. The variable is a container that contains value of any data type whether it is string, integer, boolean, list, tuple and so on. The variable is the name of the memory location where the value of the variable is stored. There is no way to declare variable in python. The variable is created in python when you assign value to the variable.

Rules for Writing Name of Variables

1. Name of the variable starts with starts with alphabets and underscore.
2. Do not start name of the variables with numbers.
3. Keywords is not considered as the name of the variable.
4. Name of the variable is either alphabetic or alphanumeric.

Assigning Value to the Variables

As we know that, when we assign value to the variable then variable is created. First we write name of the variable then we use assignment operator "=" equals to and after that we assign whatever type of value we want to assign like integer value. To assign multiple values to multiple variables respectively. Number of variable and numbers of values assigned are both equal separated by comma.

In [24]:
```
#Assigning value to variable
num = 5
word = "Ram"
isGood = False
#Assigning multivalue to multivariable
num1, num2, num3 = 1,3,5
```

I always recommend to write the name of the variable in camel case method or snake case method. In camel case method if we have the name of variables

contains more than one word then we write first letter of the first word in small letter and then first letter of other words after first word in capital letter. In snake case we underscore between the words of the variables.

In [33]:
```
#camel case style
nameOfTheTopper = "Your Name"
#snake case style
name_of_the_topper = "Your Name"
```

Comments in Python

Here, there is one more concept about the comment which you must know. The comment starts with “#” sign as it comment only single line. The interpreter ignores the comment line from the code. The comment is mainly used to describe the code to make reader or coder understand. There is one more type of comment is multiline comment which is enclosed with triple quotation marks. [“”” Hi “”””]

In [39]:
```
#single line comment
```

In [ ]:
```
#Multiline Comment
"""
Hello
Guys
"""
```

Printing Values of Variables

To print or output the value of the variable on output screen. You need print() function which is a builtin function in python. In print() function during calling you have to pass the name of the variable as an argument of the print() function. The print() function print the value of the variable it contains.

In [31]:
```
#Printing value to variable
print(num)
print(word)
print(isGood)
print(num1)
print(num2)
print(num3)
```

```
5
Ram
False
1
3
5
```

Datatypes and Its Typecasting

Here, I am using only primitive data types are as follows as:

1. Integer is a type of datatype that contains only numbers type value without any decimal points. To typecast any value in integer you should use int() function in python.
2. Float is a type of datatype that contains only numbers with decimal values. To typecast any value in float you should use float() function in python.
3. String is a type of datatype that contains only alphabetic and alphanumeric values. To type cast any value in string you should use str() function in python.
4. Boolean is a type of datatype that contains only two values "True" and "False". To typecast any value in boolean you should use bool() function in python.

In [67]:
```
x=str(56)
print(type(x))
y=int(8.42)
print(type(y))
z=float(6)
print(type(z))
```

```
<class 'str'>
<class 'int'>
<class 'float'>
```

Input Value of Variables

To input value of the variable on output screen. You need to use input() buitin funtion in python. In input() function, you just have to pass the displayed text in the screen during input in the argument of the input() function. By default, it takes any type of data in the form of string so if you want to input other type of data like integer, float, double then you have to typecast the input() function.

In [46]:
```
#By Default input()
x=input("Number:")
type(x)
```

Out[46]:
```
str
```

In [60]:
```
#By Typecasting
xInt=int(input("Integer:"))
print(type(xInt))
yFloat=float(input("Float:"))
```

```
print(type(yFloat))
```

```
<class 'int'>
<class 'float'>
```

Here, you have to use int() function to typecast the input data. First of all you have to pass input() function inside the int() function to get input in the form of integer datatype same in case of float we use float() function and in case of string you do not need to use any function. str() function is used to typecast any value in string but not use for input() function.

In [ ]:

Slicing on String

As we know that, string is an alphabetic and alphanumeric words and their combination enclosed with double qoutation marks. We can play with string with the help of the slicing concept. Slicing is a concept of extracting sub-string from the string just like there is a word Jupiter and i want to get sub-string from second letter till fifth letter. For that we use the concept of slicing.

In [14]:
```
#Slicing on String
name = "Ashwagandha"
print(name[3:7])
print(name[3:11:2])
```

```
waga
wgnh
```

After writing the name of the string variable we use square bracket both first open bracket and then close bracket. Between these two bracket we write the two position first one is starting position and after colon the end position. If you use two colon then after second colon you set number of steps it jumps. Here, the letter in seventh position is excluded but the third position letter of the word is included. Now, in third line I am using "2" as a step jumper so it jumps 2 step then print letter. The rule of slicimg says that the given position before colon is included but after colon the given position is excluded. If you not giving position at start then by default it takes 0 as starting index and if you not given end position then it takes last index of the string by default.

Slicing Rule :- [Start : End : Step]

In [17]:
```
#Slicing using Negative Indexing
job = "Scientist"
print(job[-9:-2])
#Reverse using Negative Indexing
print(job[::-1])
```

```
Scienti
tsitneicS
```

In slicing in python there is a concept of negative indexing in which we consider indexing from the end of the string along with negative sign. One more thing is that from the end of the string, the negative indexing starts from "-1" then "-2" till "-n" where n is length of the string. Here one interesting thing is that if you wnt to reverse the string you can do it with the help of negative indexing by not giving start and end position but giving the step as "-1" it jumps the string from the end till start which reverse the string.

Concatation on Strings

Here, concatation of string is the concept where we add two or more strings together to form a single string. Just like adding your first name, middle name and last name to form your full name. Here, we concate string by using one arithematic operator named **"+"** addition operator. Here, we use this operator to add two or more strings.

In [37]:
```
#Concatation of strings
start = "Bramha"
middle = "Vishnu"
end= "Mahesh"
#Concate two strings
print(start+middle)
#Concate more than two strings
print(start+middle+end)
```

```
BramhaVishnu
BramhaVishnuMahesh
```

String and Their Functions

Here, by using some functions on string we can edit the string from its original form to edited format. Just like replacing a letter instead of target letter to change the string. Just like if we have a sentence having 5 words to split these words in sentence using comma then we use split it. Just like to change the case of the string. Just like to remove whitespaces from the given string. This is called updation on string.

In [41]:
```
#Convert into uppercase
sign = "ashok chakra"
print(sign.upper())
#Convert into lowercase
sign = "ASHOK CHAKRA"
print(sign.lower())
```

```
ASHOK CHAKRA
ashok chakra
```

1. upper() function is used in the name of the string variable then use dot after that call the upper() function. It change the lowercase in the string to uppercase and if string is in uppercase then it print uppercase with no change.
2. lower() function is used on string to convert the uppercase letter in string into lowercase letter but if the letter is in lowercase then it remains same there is no change.

3. strip() function is used on string to remove white spaces at the starting of the string and at the ending of the string.
4. replace() function is used on string to replace any targeted letter into desired letter by passing both in the arguments of the replace() function. In first argument we pass the targeted letter in second argument we pass the desired letter.

In [46]:
```
word= " Bramhos my favorite "
#Remove white spaces
print(word.strip())
#Split the sentence
print(word.split(","))
#Replace B to G
print(word.replace("B","G"))
```

```
Bramhos my favorite
[' Bramhos my favorite ']
 Gramhos my favorite
```

Format Strings Concept

To embed string in any other string we use the format string. In format string, we write the small letter **f** and then we use double quotation marks "text" and between the double quotation we write the text and if we have to embed any other string in that string then we write the embed string in curly braces {embedString}. This will help simple python programs to show output with a meaningful sentence.

In [50]:
```
#F-String Code 1
name = "Amrit Keshari"
print(f"Welcome!{name}")
```

```
Welcome!Amrit Keshari
```

f"any string or text{embed string}"

In [52]:
```
#F-String Code 2
name = "Amrit Keshari"
txt = f"Author of the Book is {name}"
print(txt)
```

```
Author of the Book is Amrit Keshari
```


In [ ]:
```

```

More on Booleans

Boolean is a type of datatype that contains only True and False values. When we convert or typecast any value into boolean then we use bool() function. This bool() function either return True or False after passing any value in the argument of bool() function. If the bool() function returns **True** then the value which we pass in the argument of bool() function is .**truthy** value. If the bool() function returns **False** then the value which we pass in the argument is **falsy** value.

In [4]:
```
#Truthy and Falsy Code
print(bool("Amrit Keshari"))
print(bool(68))
print(bool(0))
print(bool(" "))
print(bool(None))
print(bool([]))
```

```
True
True
False
True
False
False
```

All the integer values, float values, string, list, tuple, dictionary, set values are truthy values as when we pass these things in bool() function then it returns **True** as an output. If we pass empty string, None, zero, empty list, empty turtle, empty dictionary, empty set then it returns **False** as an output.

Operator & Operand

Operators are the symbols which is mainly used to perform some mathematical and logical operation in our program during computation. To perform we need operators and operands. Now operands are the data in which we perform mathematical or logical operation. Just like we add two numbers we see we have two numbers as an operand and **+** sign as an additional operator.

Types of Operators

1. Arithematic Operator is mainly used to perform mathematical operation. Just like addition(**+**), substraction (**-**), multiplication **(*)** and division(**/ %**) of numbers. It is mainly used to compute or calculate the resultant value of any given equation along with values for each variable present in the equation. “%” is remainder operator that returns

remainder and “/” division operator only it return quotient only.

In [49]:
```
#Arithematic Operator
k = 48
e = 24
print(k+e)
print(k-e)
print(k*e)
print(k/e)
print(k%e)
```

```
72
24
1152
2.0
0
```

2. Logical Operator is mainly used to perform logical operation on operands. (**and**) AND operator is used in checking conditions. If we have to check two condition is true or not then we use (**and**) operator and we put and operator between two condition or expression. If both the condition returns true then it returns true otherwise it returns false. (**or**) OR operator is used to check whether any one condition or expression from both condition or expression is true or not. If any one condition or expression is true then it returns true. (**not**) NOT operator is used to change the true into false and false into true.

In [45]:
```
#Logical Operator
k = 48
e = 24
s = 33
print(k>e and e<s)
print(k<e or e<s)
print(not (k>e))
```

```
True
True
False
```

3. Relational Operator is mainly use to perform relational operation in any expression. It compares two operand thats why it is also known as comparision operator. It returns boolean value as output. (>=) greater than equal to operator is used to check whether left operand is greater than and equal to right operand or not. (<=) less than equal to operator is used to check whether left operand is lesser than and equal to right operand or not. (==) equals to operator is used to check

whether both the operands are equals or not. (>) greater than operator is used to check whether left operand is greater than right operand or not. (<) less than operator is used to check whether left operand is less than right operand or not.

In [47]:
```
#Relational Operator
k = 48
e = 24
print(k>=e)
print(k<e)
print(k==e)
print(k>e)
print(k<=e)
```

```
True
False
False
True
False
```

4. Bitwise Operator is mainly used to perform binary operations on any numbers. It perform operation in bitwise level. After performing operation, it gives number as an output. **(&)** bitwise AND operator set bit as 1 if both the bit is 1. **(|)** bitwise OR operator set bit as 1 if one of the both bit or both the bit is 1. **(^)** bitwise XOR operator set bit as 1 if any one of the bit from both the bit is 1. (<<) left shift operator is just shift the left bit by adding zeroes in rightest side of the binary. (>>) right shift operator shift the leftmost bit to the right side of the binary.

In [43]:
```
#Bitwise Operator
k = 4
e = 2
print(k & e)
print(k | e)
print(k ^ e)
print(k << 2)
print(k >> 2)
```

```
0
6
6
16
1
```

5. Identity Operator (**is**) operator is mainly used to check whether the left operand is pertaining to object to the right operand. It just check two operand if the object of both operand is same then it returns true

otherwise it returns false. (**is not**) operator is mainly used to check whether the left operand is not pertaining to the right one then it returns true. Simply means if both the values are equal then (**is**) operator returns true.

In [59]:
```
#Identity Operator
k = 2
e = 2
print(k is e)
```

```
True
```

6. Membership Operator (**in**) operator is mainly used to check whether the left operand is present in the sequence of right operand it means left operand is just a variable and right one is sequencial type variable just like list. To check whether the particular value of variable present in the list or not. If it present in the list, it returns true otherwise false.

In [64]:
```
#Membership Operator
k = [1,3,5,7]
e = 3
print(e in k)
```

```
True
```


In [ ]:
```

```

List

List is a sequential datatype that have ability to store multiple values in a single variable. List is arranged in order that's why it is ordered. The elements of list have their own index that's why it is indexed. We can add and remove elements in list and we can replace value of the list elements that's why it is changeable. List allows duplicates values because it find the elements by using index. List is in the square brackets.

In [7]:
```
#List
list1 = [2,4,1,5,5,1,2,6]
print(list1)
```

```
[2, 4, 1, 5, 5, 1, 2, 6]
```

Accessing Elements in List

To access elements in list we use index number. First of all, we have to write the name of list variable then write open square bracket and then write the index number of the value which you want to fetch and then close the square bracket using closing square bracket. To print the value pass this whole expression in the argument of print() function.

In [9]:
```
#Accessing Elements in List
klist = [1,3,5,7]
e = klist[3]
print(e)
print(klist[0])
```

```
7
1
```

Updating Elements in List

To update elements in list we use index number of element in the list. First of all, we have to write the name of the list variable then we have to open square bracket and write index number then close the square bracket and after that use (=) assignment operator to assign new value in that index position after that write value.

In [27]:
```
#Replace value in List
klist = ["Amrit","Keshari","Yaar"]
klist[2]="NeoCode"
print(klist)
#Replace value by slicing
klist = ["Amrit","Keshari","Yaar","Drone"]
```

```
klist[2:3]=["NeoCode","Hyper"]
print(klist)
```

```
['Amrit', 'Keshari', 'NeoCode']
['Amrit', 'Keshari', 'NeoCode', 'Hyper', 'Drone']
```

Adding Elements in List

To add elements in list we use some builtin function which is used in the list. **append()** function is used to add elements at the last position of the list. **extend()** function is used to add another list in present list. **insert(index,value)** function having two arguments the first one contain inserting index position and second one contain value for inserting.

In [23]:
```
#Add value at Last in List
klist = ["Amrit","Keshari","Yaar"]
klist.append("NeoCode")
print(klist)
#Add value at Position in List
klist = ["Amrit","Keshari","Yaar"]
klist.insert(0,"NeoCode")
print(klist)
#Add value at Last in List
klist = ["Amrit","Keshari","Yaar"]
alist = ["Python","OOPS","DSA"]
klist.extend(alist)
print(klist)
```

```
['Amrit', 'Keshari', 'Yaar', 'NeoCode']
['NeoCode', 'Amrit', 'Keshari', 'Yaar']
['Amrit', 'Keshari', 'Yaar', 'Python', 'OOPS', 'DSA']
```

Removing Elements in List

To remove elements in list we use some builtin function of python. **remove(value)** function removes the first element which is same as passed value in the argument. **pop(index)** function removes the element from the list that index is passed in the argument of pop() function. **pop()** function removes the last element of the list. To clear all the elements of the list you can use **clear()** function. We can also use **del** keyword before the list name with index like list1[3] to delete list elements.

In [38]:
```
#Remove value in List
alist = [1,3, 4, 6, 7, 9]
alist.pop()
print(alist)
alist.pop(1)
print(alist)
alist.remove(4)
print(alist)
```

```
alist.clear()
print(alist)
```

```
[1, 3, 4, 6, 7]
[1, 4, 6, 7]
[1, 6, 7]
[]
```

Printing Elements of List

To print the list of the elements by traversing each element one by one or iterating each element one by one with the help of loops. Some of the loops are used to print the list of elements are for loop, for-each loop and while loop.

Using while loop

In [43]:
```
#while loop
klist = [1,2,3,4,5,6,7]
i = 0
while(i<len(klist)):
    print(klist[i])
    i = i + 1
```

```
1
2
3
4
5
6
7
```

Using for loop

In [46]:
```
#for loop
klist = [2,3,4,5,6,7,8]
for i in range(0,len(klist),1):
    print(klist[i])
```

```
2
3
4
5
6
7
8
```

Using for-each loop

In [52]:
```
#for-each loop
klist = [3,4,5,6,7,8,9]
for i in klist:
    print(i)
```

3
4
5
6
7
8
9

Slicing in List

Slicing in List is the concept in which we separate a small part of list. Just like if list have **11** elements then we extract **3rd** to **7th** element as part of the list. This is called slicing in list. To extract a part of list from given list variable, we have to first write the name of the list variable then open square bracket and write the index from where you start fetching elements is called **start**, then write the index till where you want to stop fetching data is called **end**. Note **start** is included but **end** is excluded but before **end** all are included. Another thing is **step** that tell you how much jump after first element to second element and second element to third element it mean **(n-1)th element** to **nth element** how much gaps occur. Between start and end there is one colon and between end and step there is another colon.

In [4]:
```
amritList = ["Amrit", "Keshari", "Akash", "Birendra", True, 12.4, 34]
x = amritList[3:7:1]
print(x)
```

```
['Birendra', True, 12.4, 34]
```

Slicing Rule for List :- [Start : End : Step]

You can also use negative indexing in slicing but in case of positive indexing we start counting index from the start of the element in List and index start with zero but in negative indexing index start with -1 and goes upto -n. One more thing is that when we use negative indexing then we start counting index at the end of the element in List. See the program here, I am using **-1** as a step in the Tuple so it just reverse the tuple as you see.

In [9]:
```
y = amritList[-3:-9:-1]
print(y)
```

```
[True, 'Birendra', 'Akash', 'Keshari', 'Amrit']
```


In [ ]:
```

```

Tuples

Tuples are the sequential datatype that stores multiple value in a single variable just like list. But there is one difference between the list and tuple is that we can make change in list but we cannot make change in tuple that's why tuple is unchangeable. But one thing python allow in tuple is that we can unpack the values of variable and extract it. Tuples are ordered because their each elements have index number. Tuples are indexed as index for each element exist. Tuples allow duplicate values. Tuple is always enclosed with small brackets. If you make a tuple of one element only then you have to add one comma for that in the tuple otherwise it throws error.

Accessing Elements in Tuple

To access elements of tuple you first have to write the name of the tuple after that you have to use opening square bracket and write the index number of the element you want to access and then you have to close the square bracket. To print the accessed element you have to put this **tuple[index]** inside the argument of the **print()** function.

In [11]:
```
#Accessing tuple elements
keshari = (1,4,7,0,7)
a = keshari[2]
print(a)
print(keshari[2])
```

```
7
7
```

Unpacking of Elements in Tuple

First of all, unpacking of tuple is a method in which we extract the values of variable back. Just like we have **t1 = (1,2)** then we assign each variable name for particular variable **ta,tb = t1**. It is noted that number of values present in tuple is equals to the number of variables. If it is not possible then use * asterick sign before the last variable as it takes all the values inside it.

In [13]:
```
#Unpack tuple elements
amrit = (23,56,88)
n1,n2,n3 = amrit
print(n1)
print(n2)
print(n3)
```

```
23
56
88
```

In [15]:
```python
#Unpack tuple elements
keshari = (12,23,34,45,56,99)
n1,n2,*n3 = keshari
print(n1)
print(n2)
print(n3)
```

```
12
23
[34, 45, 56, 99]
```

Updating Elements in Tuple

To add element in tuple, first of all we have to convert the tuple into list. For converting tuple into list we have to use a constructor function for list that is **list()** function. It convert any datatype values into list. Then we can easily add elements in tuple. We can use **append()** function to add elements in tuple. To delete elements in tuple, you have to first convert the tuple into list using constructor function for list that is **list()** after that you can use **remove(value)** function to delete any element in tuple. Here, **remove(value)** is a builtin function for list to remove elements from list. It has one argument in which we pass the target value which we want to delete from the list. After passing any value in the argument of **remove()** function it matches the value from list if it find, it delete that value. We can also add tuple in tuple using addition operator (**+**).

In [20]:
```python
#Add tuple elements
amritTuple = ("Hi","Hello","Fine","Hey")
print(amritTuple)
amritList = list(amritTuple)
amritList.append("Rang")
amritTuple = tuple(amritList)
print(amritTuple)
```

```
('Hi', 'Hello', 'Fine', 'Hey')
('Hi', 'Hello', 'Fine', 'Hey', 'Rang')
```

In [38]:
```python
#Delete tuple elements
amritTuple = ("Lofi","Slowed","Music", "Slowed")
print(amritTuple)
amritList = list(amritTuple)
amritList.remove("Slowed")
amritTuple = tuple(amritList)
print(amritTuple)
```

```
('Lofi', 'Slowed', 'Music', 'Slowed')
('Lofi', 'Music', 'Slowed')
```

Printing Tuple Elements

To print the each elements of the tuple we have to iterate over each element of the tuple and it can be possible when we use looping statement. So, for printing the element of the tuple we use while loop, for-each loop and for loop.

Using while loop

In [26]:
```
amritTuple = (2,3,5,6,7)
i=0
while(i<len(amritTuple)):
    print(amritTuple[i])
    i = i + 1
```

```
2
3
5
6
7
```

Using for-each loop

In [30]:
```
amritTuple = (11,23,34,94)
for i in amritTuple:
    print(i)
```

```
11
23
34
94
```

Using for loop

In [33]:
```
amritTuple = (22,33,44,99)
for i in range(0,len(amritTuple),1):
    print(amritTuple[i])
```

```
22
33
44
99
```

Slicing in Tuples

Slicing in Tuple is the concept in which we separate a small part of tuple which you can say sub-tuple. Just like if tuple have **10** elements then we extract **3rd** to **7th** element as part of the tuple. This is called slicing in tuple. To extract a part of tuple from given tuple, we have to first write the name of the tuple variable then open square bracket and write the index from where you start fetching elements is

called **start**, then write the index till where you want to stop fetching data is called **end**. Note **start** is included but **end** is excluded but before **end** all are included. Another thing is **step** that tell you how much jump after first element to second element and second element to third element it mean **(n-1)th element** to **nth element** how much gaps occur. Between start and end there is one colon and between end and step there is another colon.

Slicing Rule for Tuple :- [Start : End : Step]

In [5]:
```
amritTuple = (1,2,3,4,5,6,7,8,9,10,11)
x = amritTuple[3:7:1]
```

In [7]:
```
print(x)
```

```
(4, 5, 6, 7)
```

You can also use negative indexing in slicing but in case of positive indexing we start counting index from the start of the element in Tuple and index start with zero but in negative indexing index start with -1 and goes upto -n. One more thing is that when we use negative indexing then we start counting index at the end of the element in Tuple.

In [14]:
```
y = amritTuple[-1:-7:-1]
print(y)
```

```
(11, 10, 9, 8, 7, 6)
```

Here, I am using **-1** as a step in the Tuple so it just reverse the tuple as you see.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Set

Set is a sequential datatype just like list and tuple. But it is just opposite to list as list are ordered, indexed, changeable and allow duplicates but set is unordered, unchangeable, unindexed and don't allow duplicates.

In [3]:
```
#set program / syntax
keshari = {1,4,7,0}
#set with duplicates throw error
keshari = {1,1,1,2,3,4}
#convert into set
klist = [1,2,3,4,5]
kSet = set(klist)
```

Set is not ordered because their elements has no index and without index the element cannot be in ordered form. Another thing set does not allow duplicate values because it has no index for their element so it cannot distinguish between duplicates if index exist then it distinguish two similar value by their index only.

In [6]:
```
print(kSet)
```

```
{1, 2, 3, 4, 5}
```

Accessing Elements in Set

As we know that, in set there is no indexing that's why we cannot access elements in set through indexing but we can access the elements in set by using membership operator **(in)** operator by applying conditional statement along with membership operator. You can only use of membership operator then also it run.

In [13]:
```
#Access Set Elements
kSet = {1,2,3,4,5}
val = 3
if val in kSet:
    print("Found")
else:
    print("Not Found")
print(5 in kSet)
```

```
Found
True
```

Adding Elements in Set

To add elements in set, we use **add(value)** function. It takes value which we want to add in our set as an argument. First of all we write name of the set variable then we use dot after that we use **add(value)** function and pass value to add that

element. But one more thing as we know that set is unordered so your new element will get random place anywhere in set. We can also use of **update(set)** that takes another set variable as an argument.This function add one set to another set simple means it concate one set to another set.

In [17]:
```
#Add element using add()
amritSet = {"Hello", "Bello", "Gello", "Dello"}
amritSet.add("Uiop")
print(amritSet)
```

```
{'Gello', 'Dello', 'Hello', 'Bello', 'Uiop'}
```

In [19]:
```
#Add element using update()
keshriSet = {"Qwerty","DSA","Werty"}
amritSet = {"Hello","Bello","Gello","Dello"}
keshriSet.update(amritSet)
print(keshriSet)
```

```
{'Hello', 'Bello', 'DSA', 'Qwerty', 'Werty', 'Gello', 'Dello'}
```

Removing Elements in Set

To remove elements in set, we use **remove(value)** and **discard(value)** built in function to remove elements from the set. The only difference between these two functions is that **remove(value)** function removes element if it exists in the set but if not exixt then it throws error but in case of **discard(value)** function if value exixt in set then it removes the element containing that value and if not exixt then it does not throw error. This is the good point of **discard(value)** function. So, we should use **discard(value)** function instead of **remove(value)** function. These both function takes value(element to be deleted) as an argument. First of all, we have to write set name then dot after that call the function and pass value in it for deletion. There is one more function **pop()** it does not take any argument but after using this function it deletes the random element in the set.

In [34]:
```
#Remove element using remove()
amritSet = {"Door","Utensil","Cup","Saucer"}
amritSet.remove("Door")
print(amritSet)
amritSet.discard("Utensil")
print(amritSet)
amritSet.pop()
print(amritSet)
```

```
{'Saucer', 'Cup', 'Utensil'}
{'Saucer', 'Cup'}
{'Cup'}
```

Printing Elements of Set

For printing elements of set, we use for-each loop because set is not indexed so we cannot able to use for loop and while loop instead of for-each loop. We just have to use for-each loop and membership operator (**in**) to visit each value of the set. Here, membership operator takes one element and throw in the **ith** variable and then we print that **ith** variable and this loop operate till visiting all the elements of set but it prints the elements randomly because it has no index and it is unordered in nature.

In [42]:
```
#print elements of set
keshriSet = {"Computer", "Science", "Technology", "Coding"}
for i in keshriSet:
    print(i)
```

```
Computer
Coding
Technology
Science
```

Dictionary

Dictionary is a type of sequential datatype that stores the values and data in **key-value** format in the curly braces. To write dictionary first of all, we have to begin with open curly brace and then write key then colon after that write value and the close the curly brace. This is how we write dictionary. One thing is noted that key is always in the double quotation mark and value if any datatype no matter. Dictionary is not indexed as it is indexed with their keys. Dictionary is ordered and we can easily change the element of dictionary but it does not allow duplicate elements. If you put duplicate element then it override the first value to present value you assign recently now.

In [3]:
```
amritDict = {
    "name":"Amrit",
    "surname":"keshari",
    "laptop":"dell",
    "marks":8.6,
    "books":2
}
print(amritDict)
```

```
{'name': 'Amrit', 'surname': 'keshari', 'laptop': 'dell', 'marks': 8.6, 'book
s': 2}
```

Accessing Items in Dictionary

To access items in dictionary, we just have to first write the name of the dictionaryb then open square bracket and write the key of the dictionary whose value you want to fetch and then close square bracket. The whole thing you write put it into **print()** function as an argument then it preints the value in that particular key. Another way to access dictionary items is to use **items()** function after dictionary name, it shows all the items present in the dictionary. If you want to get list of keys present in the dictionary then you have to use **keys()** function after writing dictionary name. And if you want to get list of values present in the dictionary then you have to use **values()** function after writing dictionary name.

In [30]:
```
keshDict = {
    "X":67,
    "Y":76,
    "Z":98
}
print(keshDict["Y"])
print(keshDict.items())
print(keshDict.values())
print(keshDict.keys())
```

```
76
dict_items([('X', 67), ('Y', 76), ('Z', 98)])
dict_values([67, 76, 98])
dict_keys(['X', 'Y', 'Z'])
```

Updating and Adding the Items of Dictionary

To update or change the value of the key of dictionary, we first write name of the dictionary then open square bracket and write the name of the key whose value we want to change and then close the square bracket then assign new value to it. After this just print the dictionary you will see the changes happen. If the key name is different from the dictionary keys then dictionary add the new key name and its value in the dictionary. We can also add one dictionary with other dictionary using **update(dictionary)** function that takes dictionary as an argument.

In [15]:
```
#Updationg items
stationary = {
    "book":78,
    "copy":97,
    "pen":56
}
print(stationary)
stationary["pen"]=51
print(stationary)
```

```
{'book': 78, 'copy': 97, 'pen': 56}
{'book': 78, 'copy': 97, 'pen': 51}
```

In [19]:
```
#Adding items
pen = {
    "red":4,
    "blue":6,
    "black":9
}
print(pen)
pen["green"]=21
print(pen)
```

```
{'red': 4, 'blue': 6, 'black': 9}
{'red': 4, 'blue': 6, 'black': 9, 'green': 21}
```

In [22]:
```
#Adding or concate
amritDict = {
    "A":1,
    "B":2,
    "C":3
}
keshriDict = {
    "D":4,
    "E":5
}
```

```
print(amritDict)
print(keshriDict)
amritDict.update(keshriDict)
print(amritDict)
```

```
{'A': 1, 'B': 2, 'C': 3}
{'D': 4, 'E': 5}
{'A': 1, 'B': 2, 'C': 3, 'D': 4, 'E': 5}
```

Deleting Items in Dictionary

To delete items from the dictionary, you have to use **pop(value)** function that takes key of dictionary as an argument and delete that item. Another way to delete item from the dictionary is by using **popitem()** function that deletes directly thes last key and value pair of the dictionary that is the last item of the dictionary.

In [34]:
```
#Deleting Dictionary Item
branch = {
    "Mech":34,
    "Civil":56,
    "Electrical":33,
    "Chemical":87
}
print(branch)
branch.pop("Electrical")
print(branch)
branch.popitem()
print(branch)
```

```
{'Mech': 34, 'Civil': 56, 'Electrical': 33, 'Chemical': 87}
{'Mech': 34, 'Civil': 56, 'Chemical': 87}
{'Mech': 34, 'Civil': 56}
```

Printing Dictionary Items

To print dictionary, we use for-each loop as we know that dictionary is not indexed just like a set so it does not allow any duplicate values right. We use membership operator to fetch each and every items of the dictionary. We can also take help of the **items()** function to print all the items of the dictionary, **keys()** to print all the keys of the dictionary and **values()** to print all the values of the dictionary.

In [46]:
```
dictOfAmrit = {
    "Trust":True,
    "Books":2,
    "Thinking":108,
    "Battery":86,
    "Name":"Amrit"
}
#For keys
for i in dictOfAmrit:
```

```
    print(i)
#For values
for i in dictOfAmrit:
    print(dictOfAmrit[i])
```

```
Trust
Books
Thinking
Battery
Name
True
2
108
86
Amrit
```

In [50]:
```
#Print using functions
dictOfKeshri = {
    "Opera":34,
    "Google":"Gemini",
    "Web":"Javascript",
    "Coding":True
}
#For keys
for i in dictOfKeshri.keys():
    print(i)
#For values
for i in dictOfKeshri.values():
    print(i)
#For items
for i in dictOfKeshri.items():
    print(i)
```

```
Opera
Google
Web
Coding
34
Gemini
Javascript
True
('Opera', 34)
('Google', 'Gemini')
('Web', 'Javascript')
('Coding', True)
```

Conditional Statement

Conditional statement in the programming is the way to check our program in particular condition and how to preform in that particular condition and if condition does not matches then what program have to do. This conditional statement help us in the categorizing of datas. There are some types of conditional statements are **if statement**, **if-else statement**, **if-elif-else statement** and **match statement**.

If Statement

In case of using **if statement** we only check one condition only. if that one condition is true then we perform some task otherwise nothing is happen. If statement just check the condition that is passed in the if statement if it is true then program runs only. First of all, we write the **if** keyword then write the condition after that use colon then come to next line and use indentation before writing print statement. Here, indentation is necessary below the ":" statements. Indentation means one tab or four space.

In [5]:
```
pendrive = 52
if(pendrive>50):
    print("I use it")
```

```
I use it
```

Shorthand of if statement

In [20]:
```
if(pendrive >= 45) : print("TRUE")
```

```
TRUE
```

If-Else Statement

In case of using **if-else statement** we check single statement but now we execute program when condition meets or condition return true but if condition does not meet or condition become false then we execute another program for that or printing some messages as you wish. First of all, we write the **if** keyword then write the condition we want to check and then colon and then write program in next line with indentation in the statement. After that go to next line and start with **else** keyword and then colon after that go to next line an write program for else statement with indentation.

In [15]:
```
hdd = 1000
if(hdd > 1000):
```

```
    print("I want to buy")
else:
    print("I dont want to buy")
```

```
I dont want to buy
```

Shorthand of if-else statement

In [18]:
```
print("TRUE") if (hdd == 1000) else print("FALSE")
```

```
TRUE
```

If-Elif-Else Statement

In case of using **if-elif-else** we check more than one condition and execute different programs for different conditions just like if first condition is true then if statement execute but if second or third condition is true then elif statement execute and atlast if no one condition meets then the program of else statement execute. First of all, we write **if** statement with condition then we write **elif** keyword and then write condition after that write colon then in next line write the program you want to execute when condition of **elif** statement returns true. The more the condition, the more the **elif** block use.

In [27]:
```
storage = 100
if(storage < 100):
    print("Mobile")
elif(storage == 100):
    print("Tablet")
else:
    print("Device")
```

```
Tablet
```

Shorthand of if-elif-else

In [35]:
```
print("TRUE_1") if (storage<100) else print("TRUE_2") if (storage==100) else p
```

```
TRUE_2
```

Match Statement

There is a very new concept in Python is Match Statement Concept which is just familiar with Switch Case statement in other programming language like C, C++, Java. Few years ago, there is no concept of switch case statement in python but now it is possible due to **match statement**. Now, in match statement we can check multiple conditions and perform code for multiple condition meet each. Here, case statement (1-5) check whether match statement returning their values or not just like here match statement return 5 then case with value 5 execute its code.

case _: this is a default case which we write in python like this.

In [41]:
```
count = 5
match count:
    case 1:
        print("ONE")
    case 2:
        print("TWO")
    case 3:
        print("THREE")
    case 4:
        print("FOUR")
    case 5:
        print("FIVE")
    case _:
        print("Nothing!")
```

```
FIVE
```

Looping Statements

Looping statement is a type of statement in programming as looping statement avoid the duplication of code because it provide repetation of execution code until condition is met. There are mainly two keywords used in the looping statement are **for** keyword and **while** keyword. **for** keyword is used in for-each loop and for loop statement and **while** keyword is used in while loop statement.

While Loop Statement

While loop statement number of iteration is not known. While loop excute until the condition return true as a result, when condition false as a result then loop will not execute and loop stops. Here, we have to use iterative variable which we use in while loop to iterate over each element once that iterative element is commonly know as **ith** variable in while loop. Also remember to add increament otherwise while continue repetation of code without any stopage because everytime condition is true and this is called **infinite loop**.

To write while loop first of all, we have make an iterative variable **i** then in next line we have to write **while** keyword and then we write the condition which we want to check and then use colon after that in next line write the program and after program write increament statement **i=i+1** like this atlast.

In [20]:
```
#while loop
i = 0
while (i < 11):
    print(i)
    i = i + 1
```

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

While Loop Using Break Keyword

We use **break** keyword in looping statement to stop the repetition of loop program after meeting certain condition and hit **break** keyword. When the program hit **break** keyword, the command or control goes outside the loop and loop stop

working. Let us see the example code given below:

In [22]:
```
#while using break
i = 0
while(i < 11):
    if(i==8):
        break;
    print(i)
    i = i + 1
```

```
0
1
2
3
4
5
6
7
```

While Loop Using Continue Keyword

We use **continue** keyword in looping statement to stop the repetation for particular condition it hits only not for all after hitting the condition like **break** keyword. **continue** keyword just skip the iterative program for particular condition it hits only in the looping program.

In [ ]:
```
#while using continue
i = 0
while(i < 11):
    if(i == 8):
        continue
    print(i)
    i = i + 1
```

```
0
1
2
3
4
5
6
7
```

While Loop Using Else Statement

We use **else** statement in while loop to execute the code when condition does not meet our requirement. Suppose we execute a while loop when its condition returns true then only it executes the program but when condition returns false then loop stops instead of stoping loop we just want to send message that the condition is failed so the loop end. For this we use **else** statement.

In [1]:
```
#while using else
i = 0
while(i < 11):
    print(i)
    i = i + 1
else:
    print("Loop End")
```

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

For Loop Statement

For loop statement uses **for** keyword. Here, in for loop the number of iteration of loop is known. Here, we do not need iterative variable. This loop iterate over each element once. We use either membership operator **(in)** or **range()** function in **for loop**. When we use membership operator **(in)** then this is called for-each loop but when we use **range()** function with membership operator (**in**) then it is called for loop. First of all, we write **for** keyword then we write name of the iterative variable after that we use membership operator **(in)** and after that we use **range()** function then in first argument we pass the starting value of iterative variable and in second argument we pass the end value of the iterative variable and atlast we pass the number of steps it jump means step of the loop.

In [1]:
```
#for loop
for i in range(0,11,1):
    print(i)
```

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

You can also use single argument in the **range()** fnction and the single argument value in range function is the end value of the loop. One more thing in **range()** function the end value is excluded but before that is included in the loop.

By using **break** keyword in the for loop, **break** keyword is mainly used to remove control out of the for loop when it hits the condition where break statement present. It is mainly used to stop the loop.

In [3]:
```
#for loop  using break
for i in range(0,11,1):
    if(i == 4):
        break
    print(i)
```

```
0
1
2
3
```

By using **continue** keyword in the for loop, **continue** keyword only skip the loop when condition met where continue statement present. Only for that contion the loop is skipped but after that loop will continue running till for loop condition.

In [7]:
```
#for loop using continue
for i in range(0,7,1):
    if(i == 4):
        continue
    print(i)
```

```
0
1
2
3
5
6
```

Else statement is mainly used in for loop to excute program or print message when condition does not meet our requirements or it return false. This will print the message that the loop is now going to end.

In [9]:
```
#for loop using else
for i in range(0,5,1):
    print(i)
else:
    print("End")
```

```
0
1
2
3
4
End
```

Here, we use **for-each loop** as a looping statement because just like list and tuple not all the sequential datatypes are in ordered form or having index so it is too difficult to print their values as it is unordered and unindexed. For that for-each loop is used that visit each value once and throw them to **i** variable then that **i** variable we print. This is how for-each loop works.

In [17]:
```
setAmrit = {1,"Amrit",3,True,5,6,"Keshari",9}
for i in setAmrit:
    print(i)
```

```
1
3
Amrit
5
6
Keshari
9
```

pass is also a keyword that is mainly used in conditional statement, loop statement and during defination of functions. It is used when we do not write program for particular statement just like we write if statement and write some program but for else statement we do not write anything so instead of that we use **pass** keyword.

In [22]:
```
for i in setAmrit:
    if(i==5):
        print("FIVE")
    else:
        pass
```

```
FIVE
```

Functions in Python

The concept of function comes to decrease the reusablity of code suppose after some operation or task we have to call a program again and again after some operation. Just like take an example of ATM where you input your data and information like information of your account and PIN Code then it gives you money and receipt after that at last it print message **Thanks for using our ATM**. This is all possible due to function. Here, they use first function for taking second function for processing and giving money and third function for thanking users. This will goes continue when any user come and use the ATM Machine. This is how function is used in the real world problem. Function actually takes input and process the input then after yields the output after processing.

In [6]:
```
def func():
    print("Hello Amrit Keshari")
```

Function is a line of code which is in the particular code block that starts with the keyword named **def** then we write name of the function and then use parentheses and after that ends with colon then goes to new line and write the program of the task we want to do in our computer by using indentation. After colon in next line first you have to take either four space or one tab then start writing program. By this way, we can define the function in python. This indentation shows that your code is in the code block of particular funtion and it runs only if when its particular function is called.

In [9]:
```
func()
```

```
Hello Amrit Keshari
```

To call or invoke the function, we just have to write the name of the function and after that write the parentheses if function having some parameter then put values into the argument and after that you should call or invoke but if the function having no parameter then you do not need to pass any argument during the invoking or calling a function.

Parameter Versus Arguments

Parameters are the variables of function that is passed in the parentheses of the function during defining of the function. When we define the function then we pass some variable that shows function need some input which required to perform the task of the function. Whereas arguments are the variables that is passed in the paretheses of the function but during function call or invoke. At that time of function call we pass number of arguments required for the function. Note that the

number of parameters of the particular function is same as the the number of particular or the very function arguments.

Function Without Parameter

Function without parameter is defined by writing firstly **def** keyword and after writing name of the function and then parentheses and after parentheses we put colon we do not pass any required variable for function in the parentheses of the function. After this we just write the program of the function in the function code block by following the rule of indentation. Function without parameter is mainly used in **greeting messages** to display in the computer screen.

In [19]:
```
#define function without parameter
def welcomeSir():
    print("Welcome! Dear Sir")
#call function
welcomeSir()
```

```
Welcome! Dear Sir
```

Function With Parameter

Function with parameter is defined by writing firstly **def** keyword and after that the name of the function and then write the parentheses and then we put colon. But in this function we pass some variables in the paretheses of the function that means function is taking some input to process it then it yields output as a result. Function with parameters are mainly used in the function that perform some mathematical and logical operations on input data and values. Just like calculating bank balance after crediting or debiting money for that a function with parameter is required to calculate the total amount after withdrawl or deposit of money. We can add as many parameter as we want by separating each of them with comma. If the function takes **n** number of arguments only if it has **n** numbers of parameter. It is noted that number parameters equals to number of arguments.

In [26]:
```
#define function with parameter
money = 100
def credit(x):
    print(f"Amount after credit:{money + x}")
def debit(x):
    print(f"Amount after debit:{money - x}")
#define function without parameter
def amount():
    print(f"Amount:{money}")
```

Now see the calling or invoking of function with arguments and function without

arguments

In [30]:
```
#calling function with arguments
credit(4)
debit(7)
#calling function without argument
amount()
```

```
Amount after credit:104
Amount after debit:93
Amount:100
```

Default Parameter in Function

As we all know that parameters of a function is the variable of function that is passed in the parentheses of the function during function definition. So there is a concept of default parameter that says that during function defination that variable that is passed in the parentheses of function is also defined. If someone calls the function without argument then that function variable takes the value of default parameter as argument of the function by default. We can give as many as default parameter in a function. We can combine both normal parameter and default parameter in the parentheses of the function when it is needed.

In [3]:
```
#default parameter
def myBook(author="Amrit"):
    print(f"Author:{author}")
```

In [5]:
```
#function calling with argument
myBook("Aman Gupta")
myBook("Raj Agarwal")
#function calling without argument
myBook()
```

```
Author:Aman Gupta
Author:Raj Agarwal
Author:Amrit
```

Keyword Arguments in Function

Keyword arguments is the concept in python in which when we pass arguments during function call, in the parentheses of a function then it is called positional arguments in function because its arguments are values or datas only, but when we pass arguments in the form of key value pair in the parentheses of function it means **key="value"** then that argument is called keyword argument of the function.

In [10]:
```
#define function
```

```
def amritFunction(friend):
    print(f"My friend is {friend}")
    print(f"How are you {friend}")
```

In [12]:
```
#keyword argument
amritFunction(friend="Aditya")
amritFunction(friend="Deepak")
```

```
My friend is Aditya
How are you Aditya
My friend is Deepak
How are you Deepak
```

In a function in python, we can pass both positional arguments and keyword arguments and combination of positional arguments and keyword arguments. But when you use **(*,variableName)** asterick sign comma variableName in the parentheses of the function. Then it is mandatory to use only keyword argument if you do not use keyword argument or instead of using keyword argument you use positional argument then you get error. When you use **(variableName,/)** variable name comma backslash in the parentheses of the function. Then it is mandatory to use positional argument. If you use keyword arguments instead of positional arguments then you will get error.

By using (*,variableName) Mandatory for keyword arguments

In [15]:
```
#keyword arguments only allowed
def amritFunction(*,x):
    print(f"X:{x} balls")
```

In [17]:
```
#function call with positional arguments
amritFunction(8)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[17], line 2
      1 #function call with positional arguments
----> 2 amritFunction(8)

TypeError: amritFunction() takes 0 positional arguments but 1 was given
```

See it throws error when i use positional argument in (*,)

In [19]:
```
#function call with keyword arguments
amritFunction(x=5)
```

```
X:5 balls
```

By using (variableName,/) Mandatory for positional arguments

In [23]:
```
#positional arguments only allowed
```

```
def amritFunction(y,/):
    print(f"Y:{y} Bats")
```

In [25]:
```
#function call with positional arguments
amritFunction(7)
```

```
Y:7 Bats
```

In [27]:
```
#function call with keyword arguments
amritFunction(y=9)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[27], line 2
      1 #function call with keyword arguments
----> 2 amritFunction(y=9)

TypeError: amritFunction() got some positional-only arguments passed as keyword
arguments: 'y'
```

See it throws error when i use keyword argument in (,/)

Arbitary Arguments in Function

Arbitary Arguments is the concept in python in which if we unknown about the number of parameters of the function then we use arbitary arguments. During function calling or invoking, as we do not know the number of parameter in the function then we can pass one argument, two arguments and multiple arguments as your wish number of arguments depends. During function definition, in parameter of the function **(* variableName)** single asterick sign with variable name is written which shows unknown about the number of parameters in the function. So, we can easily put one or more than one argument in the function after seeing this.

In [48]:
```
#define function
def amritFun(*a):
    print(a)
```

In [50]:
```
#calling function with arbitary arguments
print("First Call: With 3 arguments")
amritFun(23,45,90)
print("Second Call: With 2 arguments")
amritFun(65,65)
```

```
First Call: With 3 arguments
(23, 45, 90)
Second Call: With 2 arguments
(65, 65)
```

See it prints all the values that we pass as an argument during function call. But

when it comes to print all the elements one by one using print function it print all the elements and it can be possible using loop statements easily. Lets have a look.

In [55]:
```
def amritFun(*a):
    for i in a:
        print(i)
amritFun(1,2,3,4,5,6,7)
```

```
1
2
3
4
5
6
7
```

In [58]:
```
def amritFun(*a):
    print(a[0])
    print(a[1])
    print(a[2])
amritFun(1,2,3,4)
```

```
1
2
3
```

Keyword Arbitary Arguments in Function

Keyword Arbitary Argument is the concept in python especially for keyword arguments only. During function definition in python, when we pass **(** variablename)** two astericks sign with variable name that it shows number of keyword argument is unknown in this particular function. So, during function calling or invoking we pass one keyword argument, two keyword arguments and multiple keyword arguments, it is upto you that how much arguments you want to pass in this particular function.

In [63]:
```
#define function
def amritFun(**a):
    print(a)
```

In [65]:
```
#function call with 5 key argument
amritFun(x=1,y=2,z=3,w=5)
#function call with 3 key argument
amritFun(x=4,y=8,z=9)
```

```
{'x': 1, 'y': 2, 'z': 3, 'w': 5}
{'x': 4, 'y': 8, 'z': 9}
```

List as an Argument

Yes, we can pass list as an argument of the function during function call. When we pass list as an argument then we can perform any operation with the list by using loop statement inside the function code block.

In [74]:
```
#define function
def amritFun(list1):
    for i in list1:
        print(f"Element:{i}")
```

In [76]:
```
#calling function
listAmrit = [68, "Amrit", "Keshari",True]
amritFun(listAmrit)
```

```
Element:68
Element:Amrit
Element:Keshari
Element:True
```

Return & Pass Statement

As we know that, function return data or values as a output in the program. For returning datas or values we use return statement that directly return the value. We can return any value from the function by using **return** keyword. Now, about pass statement, it is mainly used when we define a function but not writing any code inside its code block in that case to execute that function we use **pass** keyword.

In [86]:
```
#define function
def firstDouble(list1):
    return list1[0]*2
```

In [90]:
```
#function call
listAmrit = [1,2,3,4,5]
x = firstDouble(listAmrit)
print(x)
```

```
2
```

Pass Statement Program

In [93]:
```
#define function
def amritFunction():
    pass
#call function
amritFunction()
```

After using pass statement the function runs easily without any error.

Lambda in Function

Lambda is a concept in python to create an anonymous function. Lambda takes multiple arguments but it takes only one expression to execute itself in the program. As we know that lambda is anonymous function so we have to create a variable and then assign lambda function on that variable so that the value as an output which lambda yields is stored in that created variable.

In [101…
```
x = lambda a,b : a+b
print(x(3,5))
```

```
8
```

To write lambda we use **lambda** keyword after that we write desired number of arguments as variable then we use colon after colon we write an expression that perform by taking these values of the argument. As lambda function is an anonymous function so the variable where lambda function is assigned, when we print that variable then we pass parameters on that variable as a function it means we treat created variable as a function.

In [104…
```
def adder(x):
    return lambda a:x+a
plusFive = adder(5)
plusTen = adder(10)
print(plusFive(25))
print(plusTen(110))
```

```
30
120
```

As you see that we can create more function with the same function. See here we create **plusFive()** and **plusTen()** both of these function using single **adder()** function. This is the way to use the lambda function as an anonymous function in python.

In [ ]:
```

```

Recursion in Python

Recursion is the concept of function in which function call itself under base condition to execute program repeatedly. There is two parts of the recursion are **base condition** and **recursive relation/function** The **base condition** is a condition when this base condition returns true then recusive function stop its repeatation and recursion end. The another thing is **recursive relation** that shows design of the recursive function means increment or decrement of values of arguments in function or any operation with the argument of the recursive function.

In [6]:
```
#define recursive function
def printAmrit(n):
    #Base Condition
    if n == 0:
        return
    print("Amrit")
    #Here I decrease the argument value
    #Recursive Relation
    printAmrit(n-1)
```

In [8]:
```
#invoking recursive function
printAmrit(5)
```

```
Amrit
Amrit
Amrit
Amrit
Amrit
```

This printing of names can be done by using loop statements but now we can do it using recursion as recursion saves our time as compared to looping statements. Therefore, I use recursion to print number of times my name. So, it is confirmed that we can use recursion in place of looping statement. This is how recursion takes place.

In [29]:
```
#define recursive function
#with (n-1) recursive relation
def amritNumber(n):
    if n==0:
        return
    amritNumber(n-1)
    print(n)
```

In [33]:
```
#invoking recursive function
amritNumber(7)
```

```
1
2
3
4
5
6
7
```

In this given program, we use **amritNumber(n-1)** recursive relation. Now, first of all when we invoke that **amritNumber(n)** function then we pass **7** in the parentheses of the function as an argument. Then, the line of control goes to base condition and check whether the condition returns true or false. If it return false then the line of control goes out of the base condition and goes to next line where it hits **amritNumber(n-1)** which is **amritNumber(7-1) = amritNumber(6)**. After calling **aamritNumber(6)** it hits first base condition then it returns false and then the line of control goes to -> **amritNumber(5).....amritNumber(0)** when line of control goes to **amritNumber(0)** then it hits the base condition and base condition return true then in if block return statement return back to **amritNumber(1)** after that it goes to next line and print **1** then it goes repeatedly till **amritNumber(7)** and print that **7** number at last.

Here, we see that firstly amritNumber(7) comes in the stack then amritNumber(6), then amritNumber(5), then amritNumber(4) till amritNumber(0). Now, amritNumber(0) pop out from stack as it return something then amritNumber(1)pop out, then amritNumber(2) pop out, then amritNumber(3) pop out and it goes till amritNumber(7). This is how the stack is working like a stack in the system.

In [25]:
```
#define recursive function #with (n+1) recursive relation
def amritNumber(n):
    if n==0:
        return
    print(n)
    amritNumber(n-1)
```

In [27]:
```
#invoking recursive function
amritNumber(7)
```

```
7
6
5
4
3
2
1
```

Recursion is just working on the principle of **LIFO** Last In First Out. It is just working like a stack.

In [ ]:

List Comprehension

List comprehension is the concept in which we use conditional statements and looping statements inside the square bracket to create a list from the given list. It means just like we have a list of natural numbers from **1** to **51** if you want to create list of **odd** numbers and **even** numbers from the **natural** number list then you have to use **list comprehension** concept. By list comprehension we can create new list with specified data or values that is filtered from the previous list.

List Comprehension For Natural Numbers (With Looping Statement)

In [17]:
```
#With Looping Statement
listNatural = [x for x in range(52)]
print(listNatural)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 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]
```

Here, first of all we open square bracket then firstly write expression whatever if you only want to store items in list then only write **x** as an expression but if you want to double the elements after getting items from the loop then you use **x*2** as an expression. now after expression we use loop statements in single line statement after that at last you use conditional statement to filter the items from the list. Atlast we close the square bracket.

List Comprehension For Odd Numbers (With Conditional Statement & Looping Statement)

In [15]:
```
#With Conditional Statement & Looping Statement
listOdd = [x for x in listNatural if x%2 !=0]
print(listOdd)
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41,
43, 45, 47, 49, 51]
```

Here, I am using for-each loop in program because we have already list so we fetch all the items from the list. Then by using conditional statement we check all the elements of that list and then it create a new list where it puts all the elements of that list that satisfies the condition of even number whether it is divisible by two or not. If it is not divisible by two then that element is put into the new list.

List Comprehension For Even Numbers (With Conditional Statement & Looping Statement)

In [19]:
```
#With Conditional Statement & Looping Statement
```

```
listEven = [x for x in listNatural if x%2 ==0]
print(listEven)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40,
42, 44, 46, 48, 50]
```

Copy the List

To copy list in python we use a builtin function named **copy()**. This **copy()** function has no paramenters so we do not need to insert argument in this function. To use **copy()** function, we have to write name of list variable which you want to copy then use dot after that call the **copy()** function means write **copy()** function. Also assign this expression in any variable so that copy of list is stored in that variable.

In [36]:
```
#copy list
copyList = listEven.copy()
print(copyList)
print(listEven)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40,
42, 44, 46, 48, 50]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40,
42, 44, 46, 48, 50]
```

See both **listEven** list and **copyList** list are same.

Nested List Comprehension

Nested List Comprehension is just similar concept of the **List Comprehension Concept**. The only difference between the list comprehension and nested list comprehension is that list comprehension is applicable only for lists but nested list comprehension is applicable for **Matrix** (the list containing number of lists each list having same number of elements). By using nested list comprehension we can easily make a list of specified elements from the **matrix**.

In [30]:
```
#With Double Looping Statement
listMatrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16],[17,18,19,20]]
listOfNumbers = [x for ilist in listMatrix for x in ilist]
print(listOfNumbers)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Now, using nested list comprehension we convert **matrix** (List of lists) into a simpler list format. Here, in nested list comprehension we use double looping statement instead of single looping statement that we use in list comprehension.

Now we are using double looping statement along with a conditional statement. Here we use double looping statement because we work on List of list that is on

matrix so for working on matrix we need double looping statement the first looping statement is used to extract all the list from the matrix and the second looping statement is used to extract all the elements of the extracted list. Then we can apply conditional statement after writing both of these looping statements.

In [40]:
```
#With Conditional Statement & Double Looping Statement
listofEvens = [x for ilist in listMatrix for x in ilist if x%2 == 0]
print(listofEvens)
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

The first looping statement is **for ilist in listMatrix** is used to extract all the lists present in the matrix then we use second looping statement just after writing first looping statement. The second looping statement is **for x in ilist** that is used to extract all the elements from the present extracted list. Now the new list is formed that contains only elements. Now atlast we use conditional statement that is **if x%2 == 0** that is used to filter all the elements as the element satisfies the condition or not. If the element satisfies the condition then it is put into the new formed list by nest list comprehension. This is how nested list comprehension works.

The del Statement

The **del statement** is mainly used to delete the whole list, tuple, set and dictionary or you can delete the items of the tuple, list, set and dictionary easily. It uses **del** keyword for deleting the things in del statement.

del statement for List

In [50]:
```
#del statement for list
amritList = ["Amrit",True,"Keshari",89]
print(amritList)
del amritList[2]
print(amritList)
```

```
['Amrit', True, 'Keshari', 89]
['Amrit', True, 89]
```

In [52]:
```
del amritList
print(amritList)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[52], line 2
      1 del amritList
----> 2 print(amritList)

NameError: name 'amritList' is not defined
```

del statement for Tuple

In [77]:
```
#del statement for tuple
amritTuple = ("Amrit",True,"Keshari",89)
print(amritTuple)
del amritTuple[2]
print(amritTuple)
```

```
('Amrit', True, 'Keshari', 89)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[77], line 4
      2 amritTuple = ("Amrit",True,"Keshari",89)
      3 print(amritTuple)
----> 4 del amritTuple[2]
      5 print(amritTuple)

TypeError: 'tuple' object doesn't support item deletion
```

Yes, it does not support item deletion of tuple but it support overall tuple deletion.

In [70]:
```
del amritTuple
```

In [72]:
```
print(amritTuple)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[72], line 1
----> 1 print(amritTuple)

NameError: name 'amritTuple' is not defined
```

del statement for Set

In [79]:
```
#del statement for set
amritSet = {"Amrit",True,"Keshari",89}
print(amritSet)
```

```
{89, 'Keshari', 'Amrit', True}
```

In [83]:
```
del amritSet
print(amritSet)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[83], line 1
----> 1 del amritSet
      2 print(amritSet)

NameError: name 'amritSet' is not defined
```

As we know that dictionary and set both are unindexed as they have no index. So, here i am not using del statement for deleting single element just like i do in list

and tuple as they are indexed.

del statement for Dictionary

In [92]:
```
#del statement for dictionary
amritDict = {
    "A":1,
    "B":2,
    "C":3,
    "D":4,
    "E":5,
    "F":6,
    "G":7
}
print(amritDict)
del amritDict["B"]
print(amritDict)
```

```
{'A': 1, 'B': 2, 'C': 3, 'D': 4, 'E': 5, 'F': 6, 'G': 7}
{'A': 1, 'C': 3, 'D': 4, 'E': 5, 'F': 6, 'G': 7}
```

In [94]:
```
del amritDict
print(amritDict)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[94], line 2
      1 del amritDict
----> 2 print(amritDict)

NameError: name 'amritDict' is not defined
```


In [ ]:

File Handling

File handling is a concept of programming in which we handle and manage the file with the help of programming. Just like writing a python code that can create a new file, delete a file and update a file. In python, file handling can be achieved by importing **os** module in python program for deletion operation only.

Some of the function that is being used to handle and the operations happen on files like creating a file, deleting a file and updating a file. These function takes name of the file and mode of opening the file as an arguments of the function.

open(fileName,mode) is a function that is mainly used to open the file from the system. **open()** function has two parameters the first parameter is for location of the file and second parameter is for mode of opening a file. Here, in file handling in python we use **with** statement because using **with statement** handlesclosing the file after operation done on the files automatically as we do not need to write **close()** function for closing files.

In [ ]:
```
amritFile = open("C:/Users/AMRIT%20KESHARI/Downloads/Basic_Python1.pdf", "rt")
```

Here, r refers to open the file in reading mode and t refers to the operation is in for text mode.

Some modes of opening a file in file handling concept of python are as follows as:

1. **r** mode open the file for reading operation in the file, but if file does not exist in your system then it throws error. So this mode is applicable only if the file exists in your computer system.
2. **a** mode open the file for updating operation means to append some data in the file. If file does not exist in your computer system then this mode helps you to create a new file in the computer system and then append the data in that new file. It will start adding content at the end of the file means it will not overwrite the content of the file.
3. **w** mode open the file for writing operation in the file, but if file does not exist in your computer system then it will create a new file in your computer system and then start writing data in the new file. It will overwrite the whole content of the file.
4. **x** mode is used to creation of file if the file you create exist in the computer system then it throws error.
5. **b** mode is mainly used to do operation in binary mode.
6. **t** mode is mainly used to do operation in text mode.

Reading File Using File Handling

In [ ]:
```
amritFile = open("C:/Users/AMRIT%20KESHARI/Downloads/Basic_Python1.pdf", "rt")
print(amritFile.read())
amritFile.close()
```

Here, **read()** function in file handling in python use to read the files which is present in the amritFile variable. If you are not using **with statement** then you have to close the file using **close()** function.

Using with statement read the file

In [ ]:
```
with open("C:/Users/AMRIT%20KESHARI/Basic_Python1.pdf", "r") as amritFile:
    print(amritFile.read())
```

As we know that, **read()** function is used to read file. The **read()** function will print the whole text but if you want to print desired number of text then you can do it. As **read()** function have a parameter where you pass number of words you want to print as an argument.

In [ ]:
```
with open("C:/Users/AMRIT%20KESHARI/Basic_Python1.pdf", "r") as amritFile:
    print(amritFile.read(7)) #first seven text
```

Using with statement read one line of the file

In [ ]:
```
with open("C:/Users/AMRIT%20KESHARI/Basic_Python1.pdf", "r") as amritFile:
    print(amritFile.readLine())
```

By using **readLine()** function we can print one line of the file, but if you want to print two or more lines of file then you have to use **n** times of **readLine()** function for printing **n** lines from the file. As we know that the **readLine()** function returns the single line from the file.

In [ ]:
```
with open("C:/Users/AMRIT%20KESHARI/Basic_Python1.pdf", "r") as amritFile:
    print(amritFile.readLine())
    print(amritFile.readLine())
    print(amritFile.readLine())
    print(amritFile.readLine())
    print(amritFile.readLine())
```

Reading whole file

For reading the whole file, we use looping statement just like for-each loop for reading whole file. First of all we open the file using with statement and open() function then we use for-each loop to traverse all the content of the file and then print it.

In [ ]:
```
with open("C:/Users/AMRIT%20KESHARI/Basic_Python1.pdf", "r") as amritFile:
    for i in amritFile:
        print(i)
```

Writing File Using File Handling

Using with statement write the file

First of all, we use with statement and invoke open() function to open file with write mode for that we use **w** as a file mode. If you want to only insert the text without overwriting the file content then you have to use **a** append mode instead of **w** write mode. Then we use the file variable and after writing file variable use dot then invoke **write()** function. In write function, we have one parameter where we pass the string or text that i want to insert in the file. So, write the string or text in the parentheses of **write()** function.

Append (It will end this content at the end of the file)

In [ ]:
```
with open("C:/Users/AMRIT%20KESHARI/Basic_Python1.pdf", "a") as amritFile:
    amritFile.write("My name is Amrit Keshari! Call me Amrit")
```

Write (It will overwrite the content of the file)

In [ ]:
```
with open("C:/Users/AMRIT%20KESHARI/Basic_Python1.pdf", "w") as amritFile:
    amritFile.write("My name is Amrit Keshari! Call me Amrit")
```

Deleting File Using File Handling

To delete file using file handling concept in python, for that we have to first import the **os** library in our python program. In **os** library, we have some methods for deleting files and folders are as follows as:

1. **remove()** function is used to remove files from the computer system. There is one parameter in **remove()** function where we pass the address of file.
2. **path.exist()** function is used to check whether file exists in the computer system or not.
3. **rmdir()** function is used to remove folders from the computer system. There is one parameter in **rmdir()** function where we pass the address of folder.

Using remove() function

In [ ]:
```
import os
```

```
os.remove("C:/Users/AMRIT%20KESHARI/Basic_Python1.pdf")
```

Using path.exist() function along with remove() function

In [ ]:
```
import os
if os.path.exists("C:/Users/AMRIT%20KESHARI/Basic_Python1.pdf"):
    os.remove("C:/Users/AMRIT%20KESHARI/Basic_Python1.pdf")
else:
    print("File not Exists")
```

Using rmdir() function to remove folder

In [ ]:
```
import os
os.rmdir("C:/Users/AMRIT%20KESHARI/Desktop")
```


In [ ]:

In [ ]:

Class & Object in Python

Class is a user-defined datatype which has both variables and methods bounded with each other and is used to bound data and methods together. Class is written by using **class** keyword in python. First of all, we write the keyword **class** then we write name of the class after that we use colon and then go to next line. In next line, follow rule of indentation and then write the variables and functions in that class code block. In class, variables are called properties or attributes of our class and functions are called behaviours of our class. Class is just like a map or template to design an object in your python program. Just you want make your own house for that you need to draw a map of your house. That map of your house is like a class and your made house is just like an object. That's why class is called logical entity whereas object is called physical entity. Each objects have their own unique attributes or data so during each object call you will see change of behaviour of the same method by changing objects.

Class Creation

Difference between method and class

Method is a block of code inside the class that is defined just like a function but **function** is a block of code that is not present inside any class it is in the program only. To call a **Method**, you need object name then dot and then call your method, but in case of function you can call it directly by function name. **Method** can only access object variables and each object has their own variables, but **functions** access the data which you pass in the argument of the function in the parentheses of the function. **Method** pass its object inside itself in its parentheses as a **self** whereas **function** only pass variables in its parentheses.

In [55]:
```
class Amrit:
    amrit = 8
    def __init__(self,a):
        print(f"Amrit:{a}")
    def func(self):
        print("Data")
```

Here, **__init__** is a constructor method. This **__init__** method is called when an object is created. **amritFun()** is a behaviour of the particular class named **Amrit**. Object is the one that can change the behaviour of the class according to itself. Just if I create 5 objects of class named **Amrit** then 5 objects behave differently by using the same class named **Amrit**. Let's have a look in the python program.

Object Creation

When you create object then memery is allocated for each objects you create. As class only define the structure but object hold the data and functions according to the class defined. Objects are created when we call the class name just like calling a function. See in the program given below.

In [24]:
```
obj1 = Amrit(1)
obj2 = Amrit(True)
obj3 = Amrit("p")
obj4 = Amrit("sam")
obj5 = Amrit(9.05)
```

```
Amrit:1
Amrit:True
Amrit:p
Amrit:sam
Amrit:9.05
```

See the behaviour of all the 5 objects

Self Keyword

self keyword is used to show or it indicates the present instance or object of the method when any method of the class is called. Let us see how **self** keyword works. Our python program calls the function by first calling a class then call its method after that it passes the object in the argument of the method of the class. That passing of argument where present object is passed is considered in our program as **self**. Actually intrepreter of python intreprete this program like ths but we write the same thing just by calling object and the method only but behind the computer it works like calling a class then mehod and then pass object in the method.

In [40]:
```
Amrit.func(obj1)
Amrit.func(obj2)
Amrit.func(obj3)
Amrit.func(obj4)
Amrit.func(obj5)
```

```
Data
Data
Data
Data
Data
```

But in reality to code to create an object then call its method for that we just have to first create an object.

In [57]:
```
obj1 = Amrit(2)
```

```
Amrit:2
```

Now we will call the method of the object for that we use first name of the object then dot after that name of the method. That's how we call the method of the class.

In [59]:
```
obj1.func()
```

```
Data
```

Now we can also access the data and variables of the class just by using object name and the dot and the name of the variables or attribute.

In [64]:
```
obj1.amrit
```

Out[64]: 8

First of all we define the class and its variable and methods inside the class scope. Let us see in the program given below. Note that follow the rule of indentation during class definition because you are going to create methods and methods comes inside the class code block. So all the methods should be inside the class code block named **AmritCalc**.

In [131…
```
class AmritCalc:
    def __init__(self):
        self.f = 6
        self.s = 9
    def add(self):
        print("Add:", self.f)
    def sub(self):
        print(f"Sub: {self.f - self.s}")
    def mul(self):
        print(f"Mul: {self.f * self.s}")
    def quo(self):
        print(f"Quo: {self.f / self.s}")
    def rem(self):
        print(f"Rem: {self.f % self.s}")
```

When we use variable **f** and **s** then we have to use **self** followed by dot.

Now we access the methods of the class by using object. For that first of all we have to create object. When we create object then f and s variable is formed due to constructor method. So for creating an object we have to call class name as a function.

In [134…
```
calcObject = AmritCalc()
```

Now we have to access the method of **calcObject** method for that we call the method by using object and dot. Lets see in the program given below.

In [137…
```
calcObject.add()
```

```
Add: 6
```

In [139…
```
calcObject.sub()
```

```
Sub: -3
```

In [141…
```
calcObject.mul()
```

```
Mul: 54
```

In [143…
```
calcObject.quo()
```

```
Quo: 0.6666666666666666
```

In [145…
```
calcObject.rem()
```

```
Rem: 6
```


In [ ]:

In [ ]:

In [ ]:

In [ ]:

_ _ init _ _(self) Method

_ _ init _ _(self) method is a constructor method which is present inside the class code block and it is the first defined method in the class code block. **_ _ init _ _(self)** method is mainly use to initialize the value to the object which you create with the help of class. So see how we assign values to the class in the given program. Also remember before writing _ _ **init** _ _ you have to write the keyword named **def** and after writing _ _ **init** _ _ you have to give parentheses and pass the self as an parameter of the constructor method.

Create a class and define _ _ **init** _ _ **(self)** in the class and then create a method in the class to access the variable of the object. Here, **self.a = a** is mainly used to assign the value to the object of the class. This will assign the value of **a** which taken from the argument is assigned to the object variable.

In [23]:
```
class Amrit:
    def __init__(self, a):
        self.a = a
    def printAmrit(self):
        print(f"value of a:{a}")
```

Create an object to access the variable of the object. That variable is assigned in the object. It can be accessed by calling the method through object.

In [19]:
```
objectAmrit = Amrit(5)
```

In [21]:
```
objectAmrit.printAmrit()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[21], line 1
----> 1 objectAmrit.printAmrit()

Cell In[16], line 5, in Amrit.printAmrit(self)
      4 def printAmrit(self):
----> 5     print(f"value of a:{a}")

NameError: name 'a' is not defined
```

As we see that we got an error. Do you know why this error comes? It comes because when we access that variable inside the method then we are not using self and as we know that the variable **a** is a object variable not a local if it is local variable then we use only **a** but if it is an object variable then we have to self keyword followed by dot before the name of the variable. Here, the error says that **a** is not defined.

Here is the corrected code

In [33]:
```
class Amrit:
    def __init__(self, a):
        self.a = a
    def printAmrit(self):
        print(f"value of a:{self.a}")
```

In [35]:
```
objectAmrit = Amrit(5)
```

In [37]:
```
objectAmrit.printAmrit()
```

```
value of a:5
```

The Another Use Of Constructor As a Greeting Message

Suppose create a code of calculator using python and you want to greet your user when they start using your code. For that you want to print **Hello User Your Most Welcome** or **Intruction before using your code for users** just like for Addition press 1 and for Multiplication press 2 like this. Then you need a constructor method of class.

See in the program how constructor method is used as a Greeting Message

In [58]:
```
class Calculator:
    def __init__(self):
        print("Welcome Sir!")
```

In [60]:
```
objectGreet = Calculator()
```

```
Welcome Sir!
```

See in the program how constructor method is used as an Instruction

In [62]:
```
class Calculator:
    def __init__(self):
        print("1 for Addition")
        print("2 for Substraction")
        print("3 for Multiplication")
        print("4 for Quotient")
        print("5 for Remainder")
```

In [64]:
```
objectInstruction = Calculator()
```

```
1 for Addition
2 for Substraction
3 for Multiplication
4 for Quotient
5 for Remainder
```

Now the question is **How many times the constructor is called when we call**

different methods of the same class?

The number of times you use the object it means you will call the function by first calling the object of the class. Just See in the program given below.

In [69]:
```
class Amrit:
    def __init__(self):
        print("Constructer is called")
    def hello(self):
        print("Hello Guys!")
```

In [71]:
```
objectFirst = Amrit()
objectSecond = Amrit()
```

```
Constructer is called
Constructer is called
```

In [73]:
```
objectFirst.hello()
objectSecond.hello()
```

```
Hello Guys!
Hello Guys!
```

When we combine both of these two couple lines of code

In [76]:
```
objectFirst = Amrit()
objectSecond = Amrit()
objectFirst.hello()
objectSecond.hello()
```

```
Constructer is called
Constructer is called
Hello Guys!
Hello Guys!
```

You will see that constructor is called two times because we made it two times.

When object is created a memory is allocated for the object. But you where the memory is allocated for the object? The memory for the object is allocated in the **heap memory**. All the objects you create from the class is allocated in the heap memory of the computer system. That heap memory provide address to the objects you create during object creation.

In [80]:
```
class Amrit:
    def __init__(self):
        print("Amrit Keshari")
```

In [82]:
```
object1 = Amrit()
```

```
Amrit Keshari
```

In [84]:
```
object2 = Amrit()
```

```
Amrit Keshari
```

Now we are going to see the address of both of these object we create now. Every time we create an object, a new object is created in the heap memory. Every time when you execute a new memory is allocated.

In [87]:
```
print(id(object1))
```

```
2270311708480
```

In [96]:
```
print(id(object1))
print(id(object2))
```

```
2270311708480
2270340870128
```

The memory is allocated to the object randomly not in an arranged way in the heap memory.

Comparing Two Objects

To compare two objects in python. You had to define a method in the class for camparing two objects in python. It was because python did not know how to compare two objects in python, but now it is possible by using assignment operator(==). See in the program how we compare two object using method.

In [102…
```
class Amrit:
    def __init__(self, a, b):
        self.a = a
        self.b = b
```

In [104…
```
object1 = Amrit(12,91)
```

In [106…
```
object2 = Amrit(56,23)
```

Now, both of these two objects are created. Here, when we call the class as a function during object creation. Then we pass only two arguments are **12** and **91** in the **object1** and **56** and **23** in the object2. But our program is explicitly or bydefault pass the present object as third argument in place of the **self** keyword.

In [115…
```
if object1.a == object2.a:
    print("Yes")
else:
    print("No")
```

```
No
```

Now, I change the value of the variable **a** of the **object1**

In [118…
```
object1.a = 56
```

In [121…
```
if object1.a == object2.a:
    print("Yes")
else:
    print("No")
```

```
Yes
```

Another way is by using another method to compare two objects of the same class

In [145…
```
class Amrit:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def compare(self, newObj):
        if self.a == newObj.a:
            return True
        else:
            return False
```

Here, the **compare()** method is defined in the class that check whether the data of two objects is equals or not. Here, in **compare(self)** in place of the self the object that uses this **compare(self)** function is passed then the object from whom to compare is passed in the place of **newObj**.

In [147…
```
objectOf1 = Amrit(11,33)
```

In [149…
```
objectOf2 = Amrit(66,44)
```

In [151…
```
if objectOf1.compare(objectOf2):
    print("Yes")
else:
    print("No")
```

```
No
```

Now after updating the value of **a** of the **objectOf1**

In [156…
```
objectOf1.a = 66
```

In [158…
```
if objectOf1.a == objectOf2.a:
    print("Yes")
else:
    print("No")
```

```
Yes
```

Now we compare the value of **b** of **objectOf1** with the value of **b** of **objectOf2**

In [171…
```
if objectOf1.b == objectOf2.b:
    print("Yes")
else:
    print("No")
```

```
No
```

We make a new update function that update the value of **b**

In [188…
```
class Amrit:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def compare(self, newObj):
        if self.a == newObj.a:
            return True
        else:
            return False
    def update(self):
        self.b = 4
```

In [190…
```
objectOf3 = Amrit(2,5)
```

In [192…
```
objectOf4 = Amrit(3,4)
```

In [194…
```
objectOf3.update()
```

This **update()** method will update the value of b in the object and assigned it to the variable of the object.

In [201…
```
if objectOf3.b == objectOf4.b:
    print("Yes")
else:
    print("No")
```

```
Yes
```

Empty Class

To create an empty class you just have to write **pass** keyword inside the class code block by following the rule of indentation.

In [204…
```
class Amrit:
    pass
```

This is the simple program of the empty class. Empty class also have their objects. And their objects also present in the heap memory. In python all are like int, float

are objects. See the proof of object of the empty class in the given program.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Varibles & Methods in OOPS

Types of Variables in OOPS

There are variables which we use in our class during class definition. There are mainly two types of variables in the class are as follows as based upon the place where they defined:

1. **Class Variable** is a type of variable in the class which is assigned outside the **_ _ init _ _ (self)** function. The **class variable** is a variable in the class which is common for all the objects you create from the same class. Just like take an example of animals so all the animals have four legs which is common for all the animals. So number of legs is considered as the class variable in real life example. When you change the **class variable** then the changes happen to all the objects you create from the same class.

Class variables are also known as Static Variable

Here is the program for the class variable given below

In [11]:
```
class Animal:
    leg = 4
    def __init__(self, color):
        self.color = color
    def printLegs(self):
        print(f"Legs:{leg}")
```

Now, we create different objects of the particular class

In [14]:
```
object1 = Animal("Black")
object2 = Animal("Green")
object3 = Animal("Red")
```

In [16]:
```
print(object1.leg)
print(object2.leg)
print(object3.leg)
```

```
4
4
4
```

The value of leg variable of all the objects we create from the same class is same and when we change the class variable named leg then the value of leg variable changes for all the objects formed from the same class.

To change the value of the class variable named **leg** we have to first write the name of the class then use dot then write the name of the class variable and use assignment operator named equals to (=) then assign value.

In [21]:
```
Animal.leg = 7
```

In [23]:
```
print(object1.leg)
print(object2.leg)
print(object3.leg)
```

```
7
7
7
```

Now, the value changes for all the objects we create from the same class.

2. **Instance Variable** is a variable that refers to the object. Instance variable is defined inside the **_ _ init _ _(self)** method in the class. Now, this is a type of variable in class which carry different value for different objects of the same class. Just like take an example of the color of the animals like the color of oxes are black and the color of cows are white as their colors are the real life example of instance variable.

In [27]:
```
class Animal:
    def __init__(self, color):
        self.color = color
    def printColor(self):
        print(f"Color:{self.color}")
```

Now create different objects of the same class

In [30]:
```
object1 = Animal("Red")
object2 = Animal("Green")
object3 = Animal("Yellow")
```

See we have assigned different values to the variable of the different objects. To access the object variables, we have to write the name of the object followed by dot then write name of the method to access the value of the different objects of the same class.

In [33]:
```
print(object1.color)
print(object2.color)
print(object3.color)
```

```
Red
Green
Yellow
```

As you have seen that different objects have different values. This is how instance variable look like.

Types of Methods in OOPS

There are some functions which we use during class definitions are called methods. There are mainly three types of methods in class in python are class method, instance method and static method. Here, class method and static method both are different but in case of the variable class variables and static variables both are same.

1. **class method** is a type of method in class which belongs to **class** not belongs to **object**. **class method** uses class variables inside the class method.To define the class method first of all we use the **decorator** named **@classmethod** this will add the **cls** keyword inside the argument of the class method just like **self** in the instance method. During define class method, we pass cls as parameter of the class method. Then we start defining the class method the only difference between class method and instance method is that in class method we use the **keyword cls** instead of keyword **self**. To access variable inside the class method we write **cls** keyword then dot after that name of the variable. To call a class method, first of all we write name of the class then we use dot after that we use name of the class method. Explicitly inside the argument of the class method, **cls** is passed in place of **cls** class is passed.

If you are not using the decorator named @classmethod , then you will get error. So, you have to use @classmethod decorator before defining the class method.

In [63]:
```
class Amrit:
    name = "Amrit"
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def printName(cls):
        print(f"Name:{cls.name}")
```

In [66]:
```
Amrit.printName()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[66], line 1
----> 1 Amrit.printName()

TypeError: Amrit.printName() missing 1 required positional argument: 'cls'
```

Now, after using the decorator named @classmethod, this will add cls as an argument in your class method

In [50]:
```
class Amrit:
    name = "Amrit"
    def __init__(self,x,y):
        self.x = x
        self.y = y
    @classmethod
    def printName(cls):
        print(f"Name:{cls.name}")
```

We do not need to create object to access the class method because it pertains to the class.

In [57]:
```
Amrit.printName()
```

```
Name:Amrit
```

2. **instance method** is a type of method in class which belongs to object and not belongs to class. **instance method** uses instance variables inside the instance method. Here in instance method we are not using any decorators just like class method and static method. Here, we use the keyword named **self** in the parameter of the instance method during definition of the instance method. Now, when we call instance method, we first need to create object then we write the name of the object then dot after we write the name of the instance method. Explicitly, inside the argument of the instance method self is passed where inplace of self the present object is passsed.

Instance method is of two types are Accessor methods and Mutator Methods are as follows as:

1. **Accessor Methods** are the types of instance method that is mainly used to access the value of the instance variable. Just like as we know that each object contains its own value of the variable. So, this method is mainly use to fetch value from the instance or object.

In [72]:
```
class Amrit:
    def __init__(self, x):
        self.x = x
    def printX(self):
        print(self.x)
```

In [74]:
```
object = Amrit(34)
```

In [76]:
```
object.printX()
```

```
34
```

2. **Mutator Methods** are the types of instance method that change the present value of the instance variable. Just like each object carry its own value of its variable. So, if you want to change its value you just need to use the instance method named **Mutator Method**.

In [79]:
```
class Amrit:
    def __init__(self):
        self.x = 7
    def updateX(self, x):
        self.x = x
```

In [81]:
```
object = Amrit()
```

In [83]:
```
object.updateX(11)
```

In [85]:
```
object.x
```

Out[85]:
```
11
```

See the value is updated when we use mutator method

3. **static method** is a type of method in class which is not using **instance variable** and **class variable**, it is very different from **class method and instance method**. To define the static method you just have to write def keyword then name of the method and no need to pass any keyword like **cls** and **self** as it is not used in static method. After that by following indentation write the program inside the function code block. One more thing is that you have to use the decorator named **@staticmethod** just before defining the static method. To access the static method first of all we use name of the class then dot after that name of the static method.

If you are not using the decorator named @staticmethod, then you will not get any error now but in past you got error.

In [109…
```
class Amrit:
    def __init__(self):
        self.x = 4
    def printMsg():
        print("Hello Indians!")
```

In [111…
```
Amrit.printMsg()
```

```
Hello Indians!
```

Here is the corrected code

In [114…
```
class Amrit:
    def __init__(self):
        self.x = 4
    @staticmethod
    def printMsg():
        print("Hello Indians!")
```

In [116…
```
Amrit.printMsg()
```

```
Hello Indians!
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Inner Class Concept

The concept of inner class concept is a methodology of object oriented programming in which we define a **class** inside another class. It means **child class** inside a **parent class**. Just like we have a class named CSE in which we can define sub-class like Artifical Intelligence, Machine Learning and Data Science.After defining inner class inside the outer class, you have to make **object of inner class** inside the **_ _ init _ _(self)** method of **outer class**.

During object creation of **inner class**, we need to first write name of the outer class then dot after that we have to call inner class just like a function. See in the program - > **self.mahi = CSE.ai()**

In [ ]:
```
class CSE:
    def __init__(self):
        self.val = 18
        self.mahi = CSE.ai()
    def printSum(self):
        print(f"Outer:{self.val}")
        self.mahi.display()
    class ai:
        def __init__(self):
            self.x = 12
        def display(self):
            print(f"Inner:{self.x}")
```

In [ ]:
```
objCSE = CSE()
```

In [ ]:
```
objCSE.printSum()
```

```
Outer:18
Inner:12
```

You can make a hierarchy of classes by using the concept of inner class in python. See in the program given below are as follows as :

In [ ]:
```
class Grand:
    def __init__(self):
        self.grandAge = 85
        self.pat = Grand.Paternal()
        self.mat = Grand.Maternal()
    def display(self):
        print(f"Grand:{self.grandAge}")
        self.pat.dispfst()
        self.mat.dispscd()
    class Paternal:
        def __init__(self):
            self.patAge = 45
            self.boy = Grand.Paternal.Son()
```

```
        def dispfst(self):
            print(f"Paternal:{self.patAge}")
            self.boy.printBoy()
        class Son:
            def __init__(self):
                self.sonAge = 17
            def printBoy(self):
                print(f"Son:{self.sonAge}")
    class Maternal:
        def __init__(self):
            self.matAge = 39
            self.girl = Grand.Maternal.Daughter()
        def dispscd(self):
            print(f"Maternal:{self.matAge}")
            self.girl.printGirl()
        class Daughter:
            def __init__(self):
                self.DaugAge = 15
            def printGirl(self):
                print(f"Daughter:{self.DaugAge}")
```

In [ ]:
```
objGrand = Grand()
```

In [ ]:
```
objGrand.display()
```

```
Grand:85
Paternal:45
Son:17
Maternal:39
Daughter:15
```

Here, we have seen that if you have outer class then inner class then inner inner class. During object creation of innermost class you have to first write outermost class then dot then outer class then dot then inner class. This is known as hierarchical way to define an object just like - > **self.boy = Grand.Paternal.Son()**.

Inheritance in Python

Inheritance is the property of object oriented programming in which we inherit one class to another so that we can access the attributes and methods one class to another class. That is a class is inherited with another class then the attributes and behaviors of an inherited class can be easily accessible with the help of child class or sub class from which all classes are inherited.

Note:

Parent Class is also known as the **super class** and **base class**.

Child Class is also known as the **sub class** and **derived class**.

1. **Single Inheritance** is a type of inheritance in which we inherit only one class. To achieve this inheritance, we need two classes the first one class is parent class or super class and another one class is child class or sub class. Here, the child class can access the attributes and methods of the parent class. First of all we have to write the name of the **derived class** then open the parentheses and write the name of the one base class (only one class which you want to inherit) and then close the parentheses and then write the colon. After that by following the rules of indentation define the attributes and methods inside the derived class.

Here is the program for the single inheritance. Please have a look at the program given below

In [ ]:
```
class Father:
    def talent1(self):
        print("Talent1 of Father")
    def skill1(self):
        print("Skill1 of Father")
```

In [ ]:
```
class Daughter(Father):
    def talent2(self):
        print("Talent2 of Daughter")
    def skill2(self):
        print("Skill2 of Daughter")
```

Now we create the object of **child class** to access the both the skills and talents of Father and Daughter.

In [ ]:
```
childClassObject = Daughter()
```

Now access all the methods of both the class using the sub class or derived class object.

In [ ]:
```
childClassObject.skill1()
```

```
Skill1 of Father
```

In [ ]:
```
childClassObject.skill2()
```

```
Skill2 of Daughter
```

In [ ]:
```
childClassObject.talent1()
```

```
Talent1 of Father
```

In [ ]:
```
childClassObject.talent2()
```

```
Talent2 of Daughter
```

2. **Multiple Inheritance** is a type of inheritance in which we inherit more than one class in the derived class or child class so that it can access the attributes and methods of more than one classes that is multiple classes. First of all we have to write the name of the **derived class** then open the parentheses and write the name of the base classes (the classes which you want to inherit) separate each class with comma and then close the parentheses and then write the colon. After that by following the rules of indentation define the attributes and methods inside the derived class.

In [ ]:
```
class Base1:
    def method1(self):
        print("Method 1 is calling")
```

In [ ]:
```
class Base2:
    def method2(self):
        print("Method 2 is calling")
```

In [ ]:
```
class Base3:
    def method3(self):
        print("Method 3 is calling")
```

Here, we are going to inherit all the base classes in the derived class.

In [ ]:
```
class Derived(Base1, Base2, Base3):
    def method4(self):
        print("Method 4 is calling")
```

Now, we are going to make the object of derived class to access the attributes and methods of all the base classes as all the base classes inherit from the derived

class.

In [ ]:
```
objectDerived = Derived()
```

Now, we access all the methods of both base classes and derived class by using object of derived class.

In [ ]:
```
objectDerived.method1()
```

```
Method 1 is calling
```

In [ ]:
```
objectDerived.method2()
```

```
Method 2 is calling
```

In [ ]:
```
objectDerived.method3()
```

```
Method 3 is calling
```

In [ ]:
```
objectDerived.method4()
```

```
Method 4 is calling
```

3. **Multilevel Inheritance** is a type of inheritance in which we inherit the **first derived class** with the **base class** and then inherit the **second derived class** from the **first derived class** then inherit the **third derived class** from the **second derived class** till last derived class we do inheritance like this. This process make a chain of levels of inheritance. Here, classes inherit from the class that is already inherit from another class which forms a hierarchical chain in inheritance.

In [ ]:
```
class A:
    def behaviour1(self):
        print("A behaviour is calling")
```

In [ ]:
```
class B(A):
    def behaviour2(self):
        print("B behaviour is calling")
```

In [ ]:
```
class C(B):
    def behaviour3(self):
        print("C behaviour is calling")
```

Now, for accessing all the methods of all the classes we need to call the object of lastly derived class.

In [ ]:
```
objectLastDerived = C()
```

Now, with the help of the object of lastly derived class we can easily access all the

methods of all the base classes that derived from other class and base class.

In [ ]:
```
objectLastDerived.behaviour1()
```

```
A behaviour is calling
```

In [ ]:
```
objectLastDerived.behaviour2()
```

```
B behaviour is calling
```

In [ ]:
```
objectLastDerived.behaviour3()
```

```
C behaviour is calling
```

4. **Hierarchical Inheritance** is a type of inheritance in which we inherit all the derived class with the same base class. Here, all the derived classes inherit the attributes and methods of the single base class. Just like your parents have four childs then all four childs has taken some features from their parents.

In [ ]:
```
class Parent:
    def parentFeature(self):
        print("Good Looking")
```

In [ ]:
```
class Son(Parent):
    def sonFeature(self):
        print("Intelligent")
```

In [ ]:
```
class Daughter(Parent):
    def daughterFeature(self):
        print("Cuteness")
```

Here, to achieve inheritance we need to create object of **n-1** number of classes where n is the number of total class used in inheritance and why n-1 because there is one base class whose object is not created to achieve inheritance. Or we can say total numbers of derived class.

To see the properties of Son

We need to create object of the class Son as it is inherited with the Parent class.

In [ ]:
```
objectSon = Son()
```

In [ ]:
```
objectSon.parentFeature()
objectSon.sonFeature()
```

```
Good Looking
Intelligent
```

We need to create object of the class Daughter as it is also inherited with the Parent class.

In [ ]:
```
objectDaughter = Daughter()
```

In [ ]:
```
objectDaughter.parentFeature()
objectDaughter.daughterFeature()
```

```
Good Looking
Cuteness
```

5. **Hybrid Inheritance** is a type of inheritance in which we combine more than one inheritance. Just combining multilevel inheritance and multiple inheritance. To combine any two or more types of inheritance is known as hybrid inheritance it means to make hybrid of different types of inheritance.

In [ ]:
```
class A:
    def call_A(self):
        print("A is calling")
```

In [ ]:
```
class D:
    def call_D(self):
        print("D is calling")
```

In [ ]:
```
class E:
    def call_E(self):
        print("E is calling")
```

Single Inheritance of class B to class A

In [ ]:
```
class B(A) :
    def call_B(self):
        print("B is calling")
```

Multiple Inheritance of class C to class B, class D, class E

In [ ]:
```
class C(B,D,E):
    def call_C(self):
        print("C is calling")
```

You can create the object of both **class B** and **class C** to achieve inheritance

In [ ]:
```
objectB = B()
objectB.call_A()
objectB.call_B()
```

```
A is calling
B is calling
```

In [ ]:
```
objectC = C()
objectC.call_B()
objectC.call_D()
objectC.call_E()
```

```
B is calling
D is calling
E is calling
```

Constructor in Inheritance

We use constructor in the class for defining values or assigning values to the variables. Here, if we use constructor in inheritance then what happens see. Suppose you create a program of simple inheritance where you inheritance a child class with a parent class. Then think about it which constructor will run first when you create an object of child class or derived class.

When you call the object of **derived class** then only the **_ _ init _ _(self)** method of derived class is called, but the **_ _ init _ _(self)** method of parent class is not called.

In [ ]:
```
class Parent:
    def __init__(self):
        print("Parent init is calling.....")
```

In [ ]:
```
class Child(Parent) :
    def __init__(self):
        print("Child init is calling.....")
```

To call the init method we have to create object, then init method will be automatically called.

In [ ]:
```
objectChild = Child()
```

```
Child init is calling.....
```

Now, if you want to call the **init** method of **Parent Class** then you have to use the **super()** and then dot and then any method of the parent class whether it is **init** method or any other method no matter.

In [ ]:
```
class Parent:
    def __init__(self):
        print("Parent init is calling.....")
```

In [ ]:
```
class Child(Parent) :
    def __init__(self):
        super().__init()
        print("Child init is calling.....")
```

Here in the code block of **Child Class**. Firstly the object of **Child Class** finds its **init** method and then run it's **init** method where it goes to first line of code block of **init** method and see that **super()** is used then **super()** is executed that calls the **init** method of **Parent Class**. After calling **init** method of **Parent class** control comes back to the **init** method of **Child class** and then execute the print() function of the **init** method.

Now, in case of multiple inheritance, there are more than one **Parent Class** in the program. Then will it call all the **init** method of all the classes or only call the **init** method of first parent class. See in the program what happens?

In [ ]:
```
class Mother:
    def __init__(self):
        print("Mother init is calling.....")
```

In [ ]:
```
class Father:
    def __init__(self):
        print("Father init is calling.....")
```

In [ ]:
```
class Child(Mother, Father):
    def __init__(self):
        super().__init__()
        print("Child init is calling.....")
```

Now, we call the object of class Child then it will execute the **init** method of class Child first then it execute the **init** method of class Mother only but not call the **init** method of Father class it is because of **Method Resolution Order** concept of Python.

Method Resolution Order

In short, Method Resolution Order can be called as MRO. Method Resolution Order is the concept in which it is order where python search for a method in a hierarchy of classes in multiple inheritance when it hits multiple classes in multiple inheritance with the similar name of the method.

The rules of the Method Resolution Order is that it start searching method from **left to right classes in parentheses** where you pass multiple classes and prioritize the left one class of the multiple inheritance where the similar method is present if not present then it goes to the second left class and then check the similar method name if exist then execute the similar method you call after super().

In [ ]:
```
class Child(Mother, Father):
    def __init__(self):
        super().__init__()
```

```
        print("Child init is calling.....")
```

Here, when the object of Child class is called then **init** method is called automatically after object creation. When **init** method of Child class is called then it hits the first line of code of the code block of **init** method and hits the **super()** then call **init** method. This line of code first search the **init** method by following the Method Resolution Order and as we know that the Method Resolution Order select the method only if it's is in the left sided class and having that similar method which we call with the super(). If no one left class having that similar method then the method of rest class where the similar method is present is called.

In [ ]:
```
objectChild = Child()
```

```
Mother init is calling.....
Child init is calling.....
```

Polymorphism in Python

Polymorphism word makes with the help of **Poly** which means many and **morph** means forms. Here, the word polymorphism means one thing can make multiple forms. Just like the sound of all the animals are different like the sound cow is different and the sound of dog is different. But both are the sound of animals.

Duck Typing

Duck Typing is the type of polymorphism methodology or a programming style where it is focused on method of object rather than the specific type of the object. **Duck Typing** follows the rule of "If it swims like a duck and quacks like a duck then it is called duck" by this principle python checks if object has any required method to complete needed operation or not rather than the specific type of the object. **Duck Typing** allow different types of objects to give response to the same method call or invoke by their own way which enhance code reusabilty.

In [ ]:
```
class Bird:
    def sound(self):
        print("Chirping")
```

In [ ]:
```
class Reptile:
    def sound(self):
        print("Hissing")
```

In [ ]:
```
class GenSound:
    def __init__(self, obj):
        print(obj.sound())
```

We can the objects of both the classes whether it is Bird or Reptile no matter because both of them have method named **sound()**.

In [ ]:
```
objBird = Bird()
objReptile = Reptile()
```

Now, we create the object of **GenSound** class and access the **sound()** method.

In [ ]:
```
objGenSound = GenSound(objBird)
```

```
Chirping
None
```

In [ ]:
```
objGenSound = GenSound(objReptile)
```

```
Hissing
None
```

This is how duck typing is working like a polymorphism in Python.

Operator Overloading

Operator overloading is the concept in which we overload the operator to do some desired operation we want. Just like the addition operator is mainly used to add integers, floats, doubles, strings but it cannot add objects of the class. But it can be possible with the help of operator overloading.

Before understanding the operator overloading, we have to understand how operators work behind the scene of the code. Actually when you write code for addition.

In [ ]:
```
a = 56
b = 96
print(a + b)
```

```
152
```

Behind the scene of the code means compiler convert **a + b** into the given code.

In [ ]:
```
a. __add__(b)
```

Out[ ]:
```
152
```

So, if you want to overload this operator then you have to override the **_ _ add _ _()** method just like to use the addition operator for adding two objects we have to override the **_ _ add _ _** method. See in the program given below.

In [ ]:
```
class Total:
    def __init__(self, mark1, mark2):
        self.mark1 = mark1
        self.mark2 = mark2
    def __add__(self, obj):
        c1 = self.mark1 + obj.mark1
        c2 = self.mark2 + obj.mark2
        c3 = Total(c1, c2)
        return c3
```

Here, in the class we redefine or override the **_ _ add _ _()** method so that it works for their objects.

In [ ]:
```
obj1 = Total(19,17)
obj2 = Total(12,15)
obj3 = obj1 + obj2
print(obj3.mark1)
print(obj3.mark2)
```

```
31
32
```

Here, the **_ _ add _ _()** method is known as the **magic method** as they are predefined in the python we only have to override these methods by our own way to achieve desired operations on desired data types. Just like add method we have all the magic methods for all the operators. So, if you need to overload any operator, you just have to overload it's magic function. The operator will be overloaded easil

If you want to see the operator overloading with another operator just like operator overloading of comparison operator or relational operator. You only have to override the **magic method** of relational operator. See in the program given below.

In [ ]:
```
class Compare:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __lt__(self, obj):
        c1 = self.x + self.y
        c2 = obj.x + obj.y
        if c1 < c2:
            return True
        else:
            return False
```

Here, we choose **<** less than relational operator to overload it so that it work for object also. For that I am overriding the magic method of less than relational operator that is **_ _ lt _ _()** magic method.

In [ ]:
```
obj1 = Compare(43,96)
obj2 = Compare(57,89)
```

In [ ]:
```
if obj1 < obj2:
    print("Obj1 is losser")
else:
    print("Obj2 is losser")
```

```
Obj1 is losser
```

Suppose if you want to print the object then you pass object as an argument inside the print function. Then print function will print its address. But we need content of the object for that we have to override the **_ _ str _ _()** magic method.

In [ ]:
```
print(obj1)
```

```
<__main__.Compare object at 0x798f5213fd90>
```

As we see when we print object of the class then it prints it's address and we need content.

In [ ]:
```
class ObjShow:
    def __init__(self, x):
        self.x = x
    def __str__(self):
        return f"{self.x}"
```

Now, we have overrided the **_ _ str _ _()**. Let us create an object of the **ObjShow** class.

In [ ]:
```
object = ObjShow(5)
```

In [ ]:
```
print(object)
```

```
5
```

Method Overloading:

Method Overloading is not possible in python because python does not support same name of the function with different parameters. And as we know that method overloading is the concept of OOPS in which we overload method by changing number of parameters in the function or by changing data types in the same function.

In [ ]:
```
class OOPS:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def sum(self, x, y):
        print(x+y)
    def sum(self, x, y, z):
        print(x+y+z)
```

In [ ]:
```
obj1 = OOPS(4,5)
obj1.sum()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-19-04840865e53b> in <cell line: 0>()
----> 1 obj1 = OOPS(4,5)
      2 obj1.sum()

NameError: name 'OOPS' is not defined
```

In [ ]:
```
obj2 = OOPS(4,6,9)
obj2.sum()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-18-a1e128e61ac7> in <cell line: 0>()
----> 1 obj2 = OOPS(4,6,9)
      2 obj2.sum()

NameError: name 'OOPS' is not defined
```

Method Overloading in Python can be achieved by using keyword arguments. See in the program given below.

In [ ]:
```
class Add:
    def sum(self, x=None, y=None, z=None):
        if x!=None and y!=None and z!=None:
            print(f"Sum: {x+y+z}")
        elif x!=None and y!=None:
            print(f"Sum: {x+y}")
        else:
            print(f"Sum:{x}")
```

Now, we create an object and then access it's **sum()** method using the object by following dot.

In [ ]:
```
obj = Add()
```

In [ ]:
```
obj.sum(8)
obj.sum(8,9)
obj.sum(5,8,9)
```

```
Sum:8
Sum: 17
Sum: 22
```

By this way you can achieve method overloading of changing of number of parameters.

Method Overriding

Method Overriding is the concept of OOPS in which a method of parent is implemented specifically in the child class which override the method of parent class in the child class. In both the classes, the name of the method, the number of parameters of the method and it's return type all are same in both parent and child class.

Single Inheritance Method Overriding

In single inheritance method Overriding, the method of parent class is overrided with the help of child class method so that the method of child class executes instead of method of parent class.

In [ ]:
```
class Parent:
    def call(self):
        print("Parent Calling.....")
```

In [ ]:
```
class Child(Parent):
    def call(self):
        print("Child Calling.....")
```

In [ ]:
```
obj = Child()
obj.call()
```

```
Child Calling.....
```

Method Overriding only happens in inheritance and types of inheritance

Multiple Inheritance Method Overriding

In Multiple Inheritance Method Overriding, we override the same method of more than one **Parent class** to the same named method of **Child class**. So that child class access the same method by their own way.

In [ ]:
```
class Mother:
    def call(self):
        print("Mother Calling...")
```

In [ ]:
```
class Father:
    def call(self):
        print("Father Calling...")
```

In [ ]:
```
class Son(Father, Mother):
    def call(self):
        print("Son Calling...")
```

Here, the derived class **Son** inherit with the class **Mother** and **Father** class. Now, the **call()** method of **Son** class override the **call()** method of their Parent classes that is **Mother** and **Father** class.

In [ ]:
```
objectChild = Son()
```

In [ ]:
```
objectChild.call()
```

```
Son Calling...
```

Multilevel Inheritance Method Overriding

In Multilevel Inheritance Method Overriding, we override the same named method in each level of inheritance to its Parent class at that level. In each level of inheritance, same named method of particular child class override the same named method of particular parent class and this happens till the last level of inheritance.

In [ ]:
```
class Grand:
    def call(self):
        print("Grand Calling...")
```

In [ ]:
```
class Parent(Grand):
    def call(self):
        print("Parent Calling...")
```

In [ ]:
```
class Child(Parent):
    def call(self):
        print("Child Calling...")
```

First of all, we call the object of **Parent** class that override the **call()** method of **Grand** class.

In [ ]:
```
objParent = Parent()
objParent.call()
```

```
Parent Calling...
```

Then, we call the object of **Child** class that override the **call()** method of **Parent** class.

In [ ]:
```
objectChild = Child()
objectChild.call()
```

```
Child Calling...
```

To override the method of parent class instead of method of child class. We use the super() function and then dot after that calling the same named method which is present in all the classes. So that in child class the method of parent class is overrided successfully

In [ ]:
```
class Parent :
    def call(self):
        print("Parent Calling...")
```

Here, we call the **call()** with super() function so that the method of parent class will execute when the object of child class is created.

In [ ]:
```
class Son(Parent):
    def call(self):
```

```
        super().call()
```

In [ ]:
```
objectChild = Son()
objectChild.call()
```

```
Parent Calling...
```

Abstract Class and Methods

Abstract class is working like a blueprint of other classes in the program. The main use of the abstract class is to define the methods inside it which can be implemented by their subclasses. The abstract class is taken from the **ABC**. We import **ABC** module from the **abc** library and then we use **abstractmethod** as **@abstractmethod**.

In [6]:
```
from abc import ABC, abstractmethod
class Engineering:
    @abstractmethod
    def nameOfBranch(self):
        pass
    def noAbstractMethod(self):
        print("This is not an abstract method")

class CSE(Engineering):
    @abstractmethod
    def nameOfBranch(self):
        print("Computer Science Engineering")

class EEE(Engineering):
    @abstractmethod
    def nameOfBranch(self):
        print("Electrical & Electronics Engineering")

class CE(Engineering):
    @abstractmethod
    def nameOfBranch(self):
        print("Civil Engineering")

class ME(Engineering):
    @abstractmethod
    def nameOfBranch(self):
        print("Mechanical Engineering")
```

Now, see the usage of the abstract method by seeing the output of the program. First of all, we have to create the object of the derived class and then call its abstract method and non-abstract method. The abstract gives different output with different objects of the same class. Let's have a look at the output of the program.

See the output of the object of the derived class CSE

In [9]:
```
obj1 = CSE()
obj1.nameOfBranch()
obj1.noAbstractMethod()
```

```
Computer Science Engineering
This is not an abstract method
```

See the output of the object of the derived class EEE

In [15]:
```
obj2 = EEE()
obj2.nameOfBranch()
obj2.noAbstractMethod()
```

```
Electrical & Electronics Engineering
This is not an abstract method
```

See the output of the object of the derived class CE

In [17]:
```
obj3 = CE()
obj3.nameOfBranch()
obj3.noAbstractMethod()
```

```
Civil Engineering
This is not an abstract method
```

See the output of the object of the derived class ME

In [19]:
```
obj4 = ME()
obj4.nameOfBranch()
obj4.noAbstractMethod()
```

```
Mechanical Engineering
This is not an abstract method
```

Here, in each object of different derived class