User Defined Functions

```
def greet():
    print("Hello students") #No return

greet()
```

```
Hello students
```

```
print(greet()) #returns None by default
```

```
Hello students
None
```

```
def greet(name):
    print("Hello ", name)

greet("Ajay")
```

```
Hello  Ajay
```

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

n = int(input("Enter a number : "))
print(square(n))
```

```
Enter a number :  2
```

```
4
```

```
# udf to generate grade based on the avg marks
# >80 - A
# >60 - B
# >50 - C
# else - F

def result(l):
    total_marks = sum(l)
    total_sub = len(l)
    avg_marks = total_marks / total_sub
    if avg_marks >= 80:
        return 'A'
    elif avg_marks >= 60:
        return 'B'
    elif avg_marks >= 50:
        return 'C'
    else:
        return 'F'

l1 = [89,75,47,100,96]
```

```
grade = result(l1)
print('Student has got :',grade)
```

```
Student has got : A
```

```
# udf to generate list of common elements between a given list & tuple

def common(l,t):
    s1 = set(l)
    s2 = set(t)
    li = list(s1 & s2)
    return li

l1= [11,22,33,44,11,22,33]
t1 = (11,22,89,44,55)
print(common(l1, t1))
```

```
[11, 44, 22]
```

```
# udf to generate list of the unique elements of both given list &
tuple

def unique(l,t):
    s1 = set(l)
    s2 = set(t)
    li = list(s1 ^ s2)
    return li

l1= [11,22,33,44,11,22,33]
t1 = (11,22,89,44,55)
print(unique(l1, t1))
```

```
[33, 55, 89]
```

```
# udf to generate dictionary of the frequency count of each element of
the list

def frequency_count(li):
    d = {}
    for i in li:
        if i not in d:
            d[i] = 1
        else:
            d[i] += 1
    return d

l1 = [11,22,33,11,11,11,22,33,44,55,66,66]
print(frequency_count(l1))
```

```
{11: 4, 22: 2, 33: 2, 44: 1, 55: 1, 66: 2}
```

Docstring

```
def add(a, b):
    """This function adds two elements"""  #docstring
    # This function adds two elements    -->comments
    return a + b

print(add.__doc__)
```

```
This function adds two elements
```

Multiple return

```
def test():
    a = 10
    b = 20
    c = 30
    return a, b, c

re = test()
print(re)
print(type(re))
```

```
(10, 20, 30)
<class 'tuple'>
```

unpacking sequence in arguments

```
def test1(a, b, c):
    print(f'a is {a}')
    print(f'b is {b}')
    print(f'c is {c}')

l1 = [10,20,30]
test1(l1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[28], line 7
      4     print(f'c is {c}')
      6 l1 = [10,20,30]
----> 7 test1(l1)

TypeError: test1() missing 2 required positional arguments: 'b' and 'c'
```

```
def test1(a, b, c):
    print(f'a is {a}')
    print(f'b is {b}')
```

```
    print(f'c is {c}')

l1 = [10,20,30]
test1(l1[0], l1[1], l1[2])

def test1(a, b, c):
    print(f'a is {a}')
    print(f'b is {b}')
    print(f'c is {c}')

l1 = [10,20,30]
test1(*l1)
```

```
a is 10
b is 20
c is 30
```

```
def test1(a, b, c):
    print(f'a is {a}')
    print(f'b is {b}')
    print(f'c is {c}')

l1 = [10,20,30,40,50,60]
test1(*l1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[33], line 7
      4     print(f'c is {c}')
      6 l1 = [10,20,30,40,50,60]
----> 7 test1(*l1)

TypeError: test1() takes 3 positional arguments but 6 were given
```

Pass by value & Pass by reference

immutable objects (int, float, string,tuple,etc.) : pass by value

```
def change(a):
    a += 10

a = 20
print(f'Before the function call :{a}')
change(a)
print(f'After the function call :{a}')
```

```
Before the function call :20
After the function call :20
```

```
def change(a):
    print(f'Id before update: {id(a)}')
    a += 10
    print(f'Id after update: {id(a)}')

a = 10
print(f'Id before function call: {id(a)}')
print(f'Before the function call :{a}')
change(a)
print(f'After the function call :{a}')
```

```
Id before function call: 140714978716376
Before the function call :10
Id before update: 140714978716376
Id after update: 140714978716696
After the function call :10
```

mutable objects (list, set, dictionary) : pass by reference

```
def change_list(l1):
    l1[0] = 100
    print(f'List Inside the function : {l1}')

l2 = [56]
change_list(l2)
print(f'List outside the function : {l2}')
```

```
List Inside the function : [100]
List outside the function : [100]
```

```
def add_list(l1):
    l1.append(10)
    print(f'List Inside the function : {l1}')

l2 = [22,33,45]
print(f'List before the function call: {l2}')
add_list(l2)
print(f'List after the function call: {l2}')
```

```
List before the function call: [22, 33, 45]
List Inside the function : [22, 33, 45, 10]
List after the function call: [22, 33, 45, 10]
```

scope of a varible

```
x = 20 #global variable
def fun():
    x = 200  #local to the fun()
    print(f'x inside the function : {x}')
```

```
print(f'x before the function call : {x}')
fun()
print(f'x after the function call : {x}')
```

```
x before the function call : 20
x inside the function : 200
x after the function call : 20
```

```
x = 20 #global variable
def fun():
    global x # to access the global variable
    x = 200  #local to the fun()
    print(f'x inside the function : {x}')

print(f'x before the function call : {x}')
fun()
print(f'x after the function call : {x}')
```

```
x before the function call : 20
x inside the function : 200
x after the function call : 200
```

lambda function

```
# udf to get square of a number
def square(x):
    return x**2

square(10)
```

```
100
```

```
# lambda function for the same:
sq = lambda x: x**2 #name is givev to this lambda function

sq(10)
```

```
100
```

```
# As lambda is anonymous function, it can be used without giving name
print((lambda x, y: x*y)(10,20))
```

```
200
```

Higher Order Functions :

Functions that are taking function as an argument

map()

It applies a given function to all the items in an input list (or any other iterable) and returns a map object (an iterator).

```
# WAP to get cube of all the elements of the list.
def cube(x):
    return x*x*x

l = [1,2,3,4,5,6]
l1 = []
for i in l:
    l1.append(cube(i))  #without using map()
print(l1)
```

```
[1, 8, 27, 64, 125, 216]
```

```
# The same can be done with using map() as follows:
l = [1,2,3,4,5,6]
l1 = list(map(cube,l))  #udf inside map()
print(l1)
```

```
[1, 8, 27, 64, 125, 216]
```

```
s1 = ['hi','hello','how','are','you']
ans = list(map(len,s1)) #inbuilt function inside map()
print(ans)
```

```
[2, 5, 3, 3, 3]
```

```
l1 = [1,2,3,4,5]
l2 = [10,10,10,10,10]
ans = list(map(lambda x,y: x*y, l1,l2)) #lambda function inside map()
print(ans)
```

```
[10, 20, 30, 40, 50]
```

```
mark = input("Enter marks of five subjects sepertaed by comma:
").split(',')
mark = list(map(int,mark))
ans = max(mark)
print(ans)
```

```
Enter marks of five subjects sepertaed by comma:  21,23,25,24,21

25
```

```
li = [1,2,3,4,5,6]
d = dict(map(lambda x: (x,x**2),li))
print(d)
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}
```

```
d1 = {1:'riya', 2:'keyur', 3:'armaan'}
d2 = dict(map(lambda i: (i[0],i[1].upper()),d1.items()))
print(d2)
```

```
{1: 'RIYA', 2: 'KEYUR', 3: 'ARMAAN'}
```

filter()

It is used to create an iterator that returns elements from the input iterable (e.g., list, tuple, etc.) for which a function returns True.

```
def is_even(n):
    return n % 2 == 0

l1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
l2 = []
for i in l1:
    if is_even(i):       #without using filter()
        l2.append(i)
print(l2)
```

```
[2, 4, 6, 8, 10]
```

```
# The same can be done using filter()
l1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
l2 = list(filter(is_even,l1))
print(l2)
```

```
[2, 4, 6, 8, 10]
```

```
s = ['nayan','keyur','jay','seema','liril']
ans = list(filter(lambda x: x == x[::-1], s))
print(ans)
```

```
['nayan', 'liril']
```

```
mark = input("Enter marks of five subjects sepertaed by comma:
").split(',')
mark = list(map(int,mark))
ans = list(filter(lambda x: x>40, mark))
print(ans)
```

```
Enter marks of five subjects sepertaed by comma:  41,23,25,43,50
```

```
[41, 43, 50]
```

```
def longer_than_four(s):
    return len(s) > 4

l1 = ["apple", "banana", "kiwi", "pear", "grape"]
ans = list(filter(longer_than_four, l1))
print(ans)
```

```
['apple', 'banana', 'grape']
```

```
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

li = [2, 3, 4, 5, 6, 7, 8, 9, 10]
ans = list(filter(is_prime, li))
print(ans)
```

```
[2, 3, 5, 7]
```

```
l1 = [11,22,33,44,11,22,36,14,11,11,11,22,2,2,22,33,33]
ans = list(set(filter(lambda x:l1.count(x)>2 and x%11==0, l1)))
print(ans)
```

```
[33, 11, 22]
```

readuce()

reduce() is a function from the functools module in Python.

It's used to apply a rolling computation to sequential pairs of values in a list.

It "reduces" a list to a single value by iteratively applying a binary function provided.

```
from functools import reduce
l = [1,2,3,4,5]
ans = reduce(lambda x, y: x+y, l)
print(ans)
```

```
15
```

```
from functools import reduce
s1 = ['hi','hello','how','are','you']
```

```
ans = reduce(lambda x, y: x+" "+y, s1)
print(ans)
```

```
hi hello how are you
```

Types of Arguments:

1. Positional Arguments (Positional - only)

2. Keyword Arguments (Keyword - only)

3. Default Arguments

4. Varible Length Positional Arguments (*args)

5. Varible Length Keyword Arguments (**kwargs)

```
def net_sal(basic, allowance, deduction):
    print("Basic is :",basic)
    print("Allowance is :",allowance)
    print("Deduction is : ",deduction)
    net = basic + allowance - deduction
    return net
```

Positional Arguments

```
net_sal(8000,6000,2000)
```

```
Basic is : 8000
Allowance is : 6000
Deduction is :  2000

12000
```

```
net_sal(2000,6000,8000)
```

```
Basic is : 2000
Allowance is : 6000
Deduction is :  8000

0
```

Keyword Arguments

```
net_sal(deduction=2000,allowance=6000,basic=8000)
```

```
Basic is : 8000
Allowance is : 6000
Deduction is :  2000

12000
```

```
net_sal(allowance=6000,8000,deduction=2000)
```

```
  Cell In[11], line 1
    net_sal(allowance=6000,8000,deduction=2000)
                                               ^
SyntaxError: positional argument follows keyword argument
```

```
net_sal(8000,allowance=6000,deduction=2000)
```

```
Basic is : 8000
Allowance is : 6000
Deduction is :  2000

12000
```

```
net_sal(8000,6000,allowance=2000)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[17], line 1
----> 1 net_sal(8000,6000,allowance=2000)

TypeError: net_sal() got multiple values for argument 'allowance'
```

```
## Order : positional --> keyword
```

Default Arguments

```
def add(a,b=0,c=0):
    return a+b+c
```

```
add(1,2,3)
```

```
6
```

```
add(1,2)
```

```
3
```

```
add(1)
```

```
1
```

```
add(3,b=3,c=9)
```

```
15
```

Positional Only Arguments

```
def add(a,b,/):
    return a+b
```

```
add(b=10,a=20)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[34], line 1
----> 1 add(b=10,a=20)

TypeError: add() got some positional-only arguments passed as keyword arguments: 'a, b'
```

```
add(10,20)
```

```
30
```

```
add(10,b=20)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[38], line 1
----> 1 add(10,b=20)

TypeError: add() got some positional-only arguments passed as keyword arguments: 'b'
```

Keyword only Arguments

```
def add(*,a,b):
    return a+b

add(a=10,b=20)
```

```
30
```

```
add(10,b=20)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[45], line 1
----> 1 add(10,b=20)

TypeError: add() takes 0 positional arguments but 1 positional argument (and 1 keyword-only argument) were given
```

```
def add(a,b,/,c,d,e,f): #first two are positional only -> rest can be
of any type
    return a+b+c+d+e+f

add(1,2,c=3,d=4,e=5,f=6)
```

```
21
```

```
add(1,2,3,4,e=5,f=6)
```

```
21
```

```
def add(a,b,/,c,d,*,e,f): #first two are positional only and last two
are keyword only, rest can be of any type
    return a+b+c+d+e+f

add(1,2,c=3,d=4,e=5,f=6)
```

```
21
```

```
add(1,2,3,4,e=5,f=6)
```

```
21
```

```
add(1,2,3,4,e=5,f=6)
```

```
21
```

Variable Length/Arbitrary Positional Argument

```
# fun() , fun(10), fun(10,20), fun(10,20,30), fun(1,2,3,.....,100)

def fun(*args): #tuple
    print(args)
    print(type(args))

fun()
```

```
()
<class 'tuple'>
```

```
fun(10)
```

```
(10,)
<class 'tuple'>
```

```
fun(10,20)
```

```
(10, 20)
<class 'tuple'>
```

```
def fun2(a,b,*args,c,d):
    print(a,b, args,c,d)
```

```
fun2()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[78], line 1
----> 1 fun2()

TypeError: fun2() missing 2 required positional arguments: 'a' and 'b'
```

```
fun2(10)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[80], line 1
----> 1 fun2(10)

TypeError: fun2() missing 1 required positional argument: 'b'
```

```
fun2(10,20,30)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[82], line 1
----> 1 fun2(10,20,30)

TypeError: fun2() missing 2 required keyword-only arguments: 'c' and 'd'
```

Variable Length/Arbitrary Keyword Argument

```
def fun3(**kwargs):
    print(kwargs)
    print(type(kwargs))
```

```
fun3()
```

```
{}
<class 'dict'>
```

```
fun3(a=10)
```

```
{'a': 10}
<class 'dict'>
```

```
fun3(a=10, b=20)
```

```
{'a': 10, 'b': 20}
<class 'dict'>

fun3(a=10, b=20,c=30)

{'a': 10, 'b': 20, 'c': 30}
<class 'dict'>

def fun4(a,b,**kwrgs):
    print(kwrgs)
    print(a)
    print(b)

fun4(10,20)

{}
10
20

fun4(10,20,x = 30)

{'x': 30}
10
20

fun4(10,20,x = 30, y=40)

{'x': 30, 'y': 40}
10
20
```