Types of Functions

---->> Based on "argument" and "return" type functions are classified into four types.

1. functions with no-arguments and no-return type.
2. functions with arguments and no-return type.
3. functions with no-arguments and return type.
4. functions with arguments and with return type.

--->> The general syntax of all functions,

```
def functionname(p1,p2,p3..):
   statement1
   statement2
   statement3
   return value1, value2,..
```

1. Functions with No-Arguments and No-return type.

Syntax:

```
def functionname():
   statement1
   statement1
   statement3
```

For example:

```
def sum():
   a = 10
   b = 20
   c = a + b
   print("Sum is : ",c)
sum()
```

2. Functions with Arguments and No-return type.

```
def functionName(p1,p2,...):
   statement1
   statement1
   statement3
```

For example:

```
def sum(a,b):
    c = a + b
    print("Sum is : ",c)
sum(10,20)
```

3. Functions with No-Arguments and with return type.

```
def functionname():
    statement1
    statement1
    statement3
    return value1, value2,...
```

Note: variable_value = functionname()

For example:

```
def sum():
    a = 10
    b = 20
    c = a + b
    return c
s = sum()
print("Sum is :", s)
```

4. Functions with arguments and with return type.

```
def functionname(p1,p2,..):
    statement1
    statement1
    statement3
    return value1, value2,...
```

For example:

```
def sum(a,b):
    c = a + b
    return c
s = sum(10,20)
print("Sum is :", s)
```

Types of arguments:-

--->> In Python we have 4 types of arguments. They are,

1. required /non-default / positional arguments.
2. default arguments.
3. keyword arguments.
4. arbitary arguments/variable length arguments. (*args **kwargs)

1: required arguments / non-default arguments:

At the time of calling function what ever the order will pass arguments values based
on that values are assigned to parameters. It is called as Positional arguments.

Example1:

```
def  f1(a,b):
     print(a+b)
f1(10,20)
```

Example2:

```
def empInfo(eid, ename, sal, dept):
    print('Employee ID is:',eid)
    print('Employee Name is:',ename)
    print('Employee Salary is:',sal)
    print('Employee Deportment is:',dept)
empInfo(101, 'Srinivas', 10000,10)
```

Output:

```
Employee ID is: 1001
Employee Name is: Srinivas
Employee Salary is: 10000
Employee Deportment is: 10
```

2. default arguments.

--->> In python at the time of declaring functions we can initialize the values to parameters. These values are called default values or default arguments.

Example1:

```
def f1(a="good morning"):
    print("hello Srinivas ", a)
f1()
f1("good evening")
```

Output:

```
hello Srinivas  good morning
hello srinivas  good evening
```

Note :

----->> If we are not giving argument values then dafault values will display.

----->>If we providing values then it returns that value.

Example2:

--->> Default parameters assume a default value if a value is not provided by the actual parameters in the function call.

```
def display_message(times,message):
      for i in range(times):
             print(message)
display_message(4 ,  'Python Srinivas')
```

Output:

```
Python Srinivas
Python Srinivas
Python Srinivas
Python Srinivas
```

--->> So we can set some default values to the formal parameters in the function definition. Those are called default arguments.

--->> So that if we don’t specify actual parameters in the function call then interpreter takes formal parameters values and continue the operation.

Example3:

```
def display_message(times = 5 ,  message = "This is Python time"):
      for i in range(times):
            print(message)
display_message()
```

Output:

```
This is Python time
This is Python time
This is Python time
This is Python time
This is Python time
```

---->> In the above function we didn't pass the actual parameters in the function call
so interpreter has taken the default values and continued the operation.
---->> If we pass the actual values when we have default values already in the function definition, then interpreter takes actual values and continue the operation.

Example4:

```
def display_message(times = 5, message = "This is Python time"):
    for i in range(times):
        print(message)
display_message(2,'Python Srinivas')
```

Output:

```
Python Srinivas
Python Srinivas
```

---->> Generally the first actual parameter will map to the first formal parameter and second actual parameters will map to the second formal parameters and so on…

Note : If we give those mappings in the reverse way then it will throw error like,

Example5:

```
def display_message(times = 5 ,  message  =  "This is Python time"):
        for  i  in range(times):
                print(message)
display_message('Narayana'  ,  3)
```

Output: TypeError: 'str' object cannot be interpreted as an integer

---->> In the above case, we can specify the parameters names while passing the value in   the function call.

---->> If we specify those names in the function call then those are called keyword arguments .

Testing with both required & default arguments :

Example1:

```
def   f1(a, b="good morning"):
        print("hello ",  a , b)
f1("Srinivas")
f1("Sri ","good evening")
```

note:- after default arguments we are not allowed to declare non-default arguments.

Example:-

```
def   f1(a="srinivas", b):
    SyntaxError:-
```

3. keyword arguments:-

A keyword argument in a function call identifies the argument by a formal parameter name.

The python interpreter is then able to use these keywords to connect the values with formal parameters.

--->> At the time of calling function we can use parameter names as keywords and we can   call in any order.

Example1:

```
def empInfo(eid, ename, sal, dept):
       print('Employee ID is:',eid)
       print('Employee Name is:',ename)
    print('Employee Salary is:',sal)
    print('Employee Deportment is:',dept)
empInfo(ename='Srinivas', eid=1001, dept=10, sal=10000)
```

Output:

```
Employee ID is: 1001
Employee Name is: Srinivas
Employee Salary is: 10000
Employee Deportment is: 10
```

Example2:

```
def display_message(times=5, message="This is Python time"):
    for i in range(times):
        print(message)
display_message(message='Python Srinivas' , times=2)
```

Output:

```
Python Srinivas
Python Srinivas
```

Example3:

```
def f1(name, msg):
      print ("hello", name,msg)
f1(name = "ram", msg = "how are you")
f1('ravi',  msg='how is it')
```

----->> here, order of arguments not a problem.

Output:-

```
hello ram how are you
hello ravi how is it
```

Error

```
f1(msg='good', 'sam')
```

SyntaxError: positional argument follows keyword argument

4: Arbitary arguments / Variable arguments

--->>> Sometimes, we do not know in advance the number of arguments that will be passed into a function. To handle this kind of situation, we can use arbitrary arguments in Python.

---->> Arbitrary arguments allow us to pass a varying number of values during a function call.

--->> We use an asterisk (*) before the parameter name to denote this kind of argument.

For example:

```
def functionName(*parameter):
      pass
```

functionName(arg1,arg2,...argN)

--->> *args and **kwargs are used in function definitions to pass a variable number of arguments to a function.
--->> The single asterisk form (*args) is used to pass a non-keyworded, variable-length argument list,
--->> The double asterisk form (**kwargs) is used to pass a keyworded, variable-length argument list.
Here is an example of how to use the non-keyworded form.

Q. This example passes one formal (positional) argument, and two more variable length arguments.
Note : The general function contains a formal (positional) argument, non-keyworded argument and keyworded argument.
--->> The syntax of a function is like ,

```
some_func (formal_args , *args , **kwargs) :
       pass
```

Q. Write a program to find sum of multiple numbers ?

```
def find_sum(*numbers):
   result = 0
   for num in numbers:
      result = result + num
   print("Sum = ", result)
# function call with 3 arguments
find_sum(1, 2, 3)

# function call with 2 arguments
find_sum(4, 9)
```

Output:

```
Sum =  6
Sum =  13
```

Variable length non-keyworded arguments,
Let’s an example of using one formal and multiple variable length non-keyworded arguments,

```
def multi_args(a,*x):
   print("Formal arg is:",a)
   for i in x:
        print("The non_keywarded arg is:",i)
   return
multi_args(10,20,'Srinivas','Python')
```

Output:

```
Formal arg is: 10
The non_keywarded arg is: 20
The non_keywarded arg is: Srinivas
The non_keywarded arg is: Python
```

Using *args in calling function

Example1:

```
def multi_args(a,*x):
   print("Formal arg is:",a)
   for i in x:
      print("The non_keywarded arg is:",i)
   return
tup1=(100,'Py','Sai')           #creating a tuple with multiple args
multi_args(10,*tup1)            #using tuple in the function call as nonkeyworded arg.
```

Output:

```
Formal arg is: 10
The non_keywarded arg is: 100
The non_keywarded arg is: Py
The non_keywarded arg is: Sai
```

Variable length keyworded arguments

Let’s an example of using one formal and multiple variable length keyworded arguments,

Example1:

```
def mul_kwargs(a,**x):
    print("The formal arg is: ",a)
    for  i  in  x:
        print(f"Another keyworded arg is: {i}: {x[i]}")
        #print("Another keyworded arg is: {}: {}".format(i, x[i]))
        #print("Another keyworded arg is: %s: %s" % (i, x[i]))
mul_kwargs(a=10,b=20,c=30)
```

Output:

```
The formal arg is:  10
Another keyworded arg is: b: 20
Another keyworded arg is: c: 30
```

Using **kwarg in the function call

Example2:

```
def mul_kwargs(a,**x):
    print("The formal arg is: ",a)
    for i in x:
        print("Another keyworded arg is: %s: %s" % (i,x[i]))
dict = {"arg1":1,"arg2":2,"arg3":"Sai"}
mul_kwargs(a=10,**dict)
```

Output:

```
The formal arg is:  10
Another keyworded arg is: arg1: 1
Another keyworded arg is: arg2: 2
Another keyworded arg is: arg3: Sai
```

----->> here we can pass any no.of arguments in place of *.

Example3:

```
def greet(*names);
    print(names)
greet(10,20,30)   ------>>   #   (10,20,30)

def   greet(**names):
        print(name)
def  fun( *args ,  **kwargs):
        pass
```

Q) what is difference between *args and **kwargs ?

All types mixing

Example1:

```
def all-val(a, b=10, *c, **d):
        print(a,b,c,d)
all-val(1,2,3,'a','f',true,x=10,y=20)
```

Output:- 1 2 (3,'a','f',true) {'x':10,'y':20}

Example2:

```
def addingval(a,*b):
     print(a,b)
addingval(10,20,30,'d',40)
```

Output : 10 (20, 30, 'd', 40)

Example3:

```
def addingval(a,**b):
     print(a,b)
addingval(a=10,b=20,c=30,d=40)
```

Output : 10 {'b': 20, 'c': 30, 'd': 40}

Example4:

```
def av(a,b=10,*c,**d) :
     print(a,b,c,d)
av(1,2,'a1',3,'f',4,5)
```

Output : 1 2 ('a1', 3, 'f', 4, 5) {}

Example5:

```
def av(a,b=10, *c,**d):
     print(a,b,c,d)
av(1,2, a1=3,c='f',d=4,r=5)
```

Output : 1 2 () {'a1': 3, 'c': 'f', 'd': 4, 'r': 5}

Example6:

```
def av(a,b=10,*c,**d):
      print(a,b,c,d)
av(1,2,3,'a','f',True,x=10,y=20)
Output : 1  2 (3, 'a', 'f', True) {'x': 10, 'y': 20}
```