1. What is the result of the code, and why?

>>> def func(a, b=6, c=8):

print(a, b, c)

>>> func(1, 2)

**Ans:-** The code will result in the following output:- 128

The function ‘func’ has three parameters, ‘a’, ‘b’, and ‘c’. The parameter ‘b’ has a default value of 6, and the parameter ‘c’ has the default value of 8. When the function is called with ‘func(1,2)’ . The argument ‘1’ is passed for the parameter ‘a’ and the argument ‘2’ is passed for the parameter ‘b’. Since a value is provided for parameter ‘b’, it overrides the default value of 6. The value of ‘c’ remains unchanged and retains its default value of , since no argument is passed for it. Therefore when the function is executed, it will print ‘1’ for ‘a’, ‘2’ for ‘b’, and ‘’8’ for ‘c’.

2. What is the result of this code, and why?

>>> def func(a, b, c=5):

print(a, b, c)

>>> func(1, c=3, b=2)

**Ans:-** The code will result in the following output :- 123

The function ‘func’ has three parameters, ‘a’, ‘b’, and ‘c’. The parameter ‘c’ has the default value of 5. When the function is called with ‘func(1, c=3, b=2)’. The argument ‘1’ passed for the parameter ‘a’, the argument ‘2’ is passed for the parameter ‘b’, and the argument ‘3’ is passed for the parameter ‘c’. In python, when calling a function and providing arguments using the ‘parameter = value’ syntax, the order of the argument doesn’t matter as long as the parameter names are specified ,this is known as keyword arguments. In this case, argument ‘2’ is explicitly assigned to the parameter ‘b’, the argument ‘3’ is explicitly assigned to the parameter ‘c’. Therefore when the function is executed, it will print ‘1’ for ‘a’, ‘2’ for ‘b’, and ‘3’ for ‘c’.

3. How about this code: what is its result, and why?

>>> def func(a, \*pargs):

print(a, pargs)

>>> func(1, 2, 3)

**Ans:-** The code will result in the following output:- 1(2,3)

The function ‘func’ has two parameters, ‘a’ and ‘\*pargs’, the parameter ‘a’ represents a regular parameter, and ‘\*pargs’ represents a variable length argument(also known as ‘packing’ or ‘tuple packing’). When the function is called with ‘func(1,2,3)’, the argument ‘1’ is passed for the parameter ‘a’, and the arguments ‘2’ and ‘3’ are passed as additional arguments. Since ‘\*pargs’ is used to collect any extra positional arguments ‘2’ and ‘3’ are packed into a tuple. Therefore when the function is executed, it will print ‘1’ for ‘a’, and ‘(2,3)’ for ‘\*pargs’, indicating that ‘2’ and ‘3’ are collected as a tuple.

4. What does this code print, and why?

>>> def func(a, \*\*kargs):

print(a, kargs)

>>> func(a=1, c=3, b=2)

**Ans:-** The code will result in the following output:- 1{‘c’:3, ‘b’:2}

The function ‘func’ has two parameters, ‘a’ and ‘\*\*kargs’. The parameter ‘a’ represents a regular parameter, and ‘\*\*kargs’ represents a variable length keyword argument parameter(also known as ‘keyword argument packing’ or ‘dictionary packing’). When the function is called with ‘func(a=1, c=3, b=2)’, the argument ‘1’ is passed for the parameter ‘a’, the keyword argument ‘c=3’ and ‘b=2’ are passed as additional arguments. Since ‘\*\*kargs’ is used to collect any extra keyword argument, the keyword arguments are packed into a dictionary, therefore,when the function is executed, ‘1’ for ‘a’ and ‘{‘c’:3, ‘b’:2}’, for ‘\*\*kargs’. indicating that the keyword argument ‘c’=3 and ‘b’=2 are collected as dictionary.

5. What gets printed by this, and explain?

>>> def func(a, b, c=8, d=5): print(a, b, c, d)

>>> func(1, \*(5, 6))

**Ans:-** The code will print the following output:- 1565

The function ‘func’ has four parameters ‘a’, ‘b’, ‘c’ and ‘d’. The parameters ‘c’ and ‘d’ have the default value of ‘c’ and ‘d’ respectively. When the function is called with ‘func(1, \*(5,6))’, the argument ‘1’ is passed for the parameter ‘a’, the tuple ‘(5,6)’ is unpacked using the ‘\*’ operator and its elements ‘5’ and ‘6’ are passed as separate arguments. The first argument ‘5’ is assigned to the parameter ‘b’, and the second argument ‘6’ is assigned to the parameter ‘c’. Since no values is provided for the parameter ‘d’, it retains its default value of ‘5’. Therefore, when the function is executed, it will print ‘1’ for ‘a’, ‘5’ for ‘b’, ‘6’ for ‘c’ and ‘5’ for ‘d’.

6. what is the result of this, and explain?

>>> def func(a, b, c): a = 2; b[0] = 'x'; c['a'] = 'y'

>>> l=1; m=[1]; n={'a':0}

>>> func(l, m, n)

>>> l, m, n

**Ans:-** The result of the code will be: (1, [‘x’], {‘a’:’y’})

In the code, a function named ‘func’ is defined with three parameters ‘a’, ‘b’ and ‘c’. Inside the function, the values of these parameters are modified. When the function is called with ‘func(‘l’, ‘m’, ‘n’)’, the value of the variable ‘l’ is passed as an argument for parameter ‘a’, the value of the variable ‘m’(which is a list[‘1’]) is passed as an argument for parameter ‘b’ and the value of the variable ‘n’ (which is a dictionary ‘{‘a’ : 0}’) is passed as an argument for parameter ‘c’.

Inside the function, the following modifications are made.

1. The parameter ‘a’ is assigned the value of ‘2’. However since ‘a’ is a local variable within the function, this assignment doesn’t affect the original variable ’l’.
2. The first element of the list ‘b’ is modified to ‘x’. Since ‘b’ is a mutable object. (a list), this modification affects the original list ‘m’ as well.
3. The value ‘a’ associated with the key ‘a’ in the dictionary ‘c’ is modified to ‘y’. Similar to the list, since ‘c’ is the mutable object(a dictionary). This modification affects the original dictionary ‘n’.

Therefore, after executing the code, the values of the variable ‘l’, ‘m’, and ‘n’ are :

1. ‘l’ remains unchanged and retains its value of 1.
2. ‘m’ is modified and its first element is now ‘x’.
3. ‘n’ is modified and the value associated with the key ‘a’ is now ‘y’.

Hence, when the expression’l’, ‘m’, ‘n’ is evaluated, it will return ‘(1, [‘x’], {‘a’: ‘y’})’.