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

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

print(a, b, c)

>>> func(1, 2)

**OutPut :**

**1 2 8**

The code you provided defines a function named func with three parameters: a, b, and c. The parameters b and c have default values of 6 and 8, respectively.

When you call the function func(1, 2), you provide arguments for the parameters a and b, but not for c. In this case, the function will use the default value of c, which is 8.

The value of a is 1, the value of b is 2 (provided as an argument), and the value of c is 8

**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)

**OutPut :**

**1 2 8**

The code you provided defines a function named func with three parameters: a, b, and c. The parameter c has a default value of 5.

When you call the function func(1, c=3, b=2), you provide arguments for all three parameters a, b, and c. The arguments are passed using keyword arguments, where the parameter names are specified explicitly.

The value of a is 1, the value of b is 2 (specified as b=2), and the value of c is 3

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

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

print(a, pargs)

>>> func(1, 2, 3)

**OutPut :**

**1 (2, 3)**

**4. What does this code print, and why?**

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

print(a, kargs)

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

**OutPut :**

**1 {'c': 3, 'b': 2}**

The code you provided defines a function named func with a parameter a and a special parameter \*\*kwargs. The \*\*kwargs parameter allows you to pass multiple keyword arguments to the function, which are then collected into a dictionary.

When you call the function func(a=1, c=3, b=2), you provide keyword arguments a=1, c=3, and b=2. These arguments are collected into the kwargs dictionary.

The value of a is 1, and the kwargs dictionary contains the key-value pairs {'c': 3, 'b': 2}, where the keys are the parameter names and the values are the corresponding argument values.

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))

**OutPut :**

**1 5 6 5**

the function func is defined with four parameters: a, b, c, and d. The parameters c and d have default values of 8 and 5, respectively.

When you call the function func(1, \*(5, 6)), you pass two positional arguments: 1 and the tuple (5, 6). The asterisk (\*) before (5, 6) unpacks the tuple, so it is equivalent to calling func(1, 5, 6).

The value of a is 1, the value of b is 5, the value of c is 6 (provided as the second element of the unpacked tuple), and the value of d is 5

**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

**Output :**

**l: 1**

**m: ['x']**

**n: {'a': 'y'}**

func that takes three parameters: a, b, and c. Inside the function, a is reassigned the value 2, b[0] is modified to 'x', and c['a'] is modified to 'y'.

After defining the function, you create three variables: l with a value of 1, m as a list with a single element 1, and n as a dictionary with a key 'a' and a value of 0.

When you call the function func(l, m, n), the values of l, m, and n are passed as arguments to the function. The function modifies a, b, and c inside the function scope, but it does not affect the original variables outside the function.

The value of l remains unchanged (still 1) because it is passed by value. The value of m changes to ['x'] because it is passed by reference and the function modifies the first element. The value of n changes to {'a': 'y'} because it is passed by reference and the function modifies the value associated with the key 'a'.