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

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

print(a, b, c)

>>> func(1, 2)

1 2 8

1. The function func is defined with three parameters: a, b, and c. The parameter b has a default value of 6, and the parameter c has a default value of 8.
2. When the function is called as func(1, 2), the value 1 is assigned to the parameter a, and the value 2 is assigned to the parameter b. Since no value is provided for the parameter c, it takes its default value of 8.
3. Therefore, when the print statement inside the function is executed, it will output 1 2 8, indicating the values of a, b, and c respectively.

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)

1 2 3

* The function func is defined with three parameters: a, b, and c. The parameter c has a default value of 5.
* When the function is called as func(1, c=3, b=2), the value 1 is assigned to the parameter a, the value 2 is explicitly assigned to the parameter b, and the value 3 is explicitly assigned to the parameter c.
* The order of the arguments in the function call does not matter when using keyword arguments. By specifying c=3 and b=2, we are explicitly assigning values to the respective parameters, overriding the default value of c.
* Therefore, when the print statement inside the function is executed, it will output 1 2 3, indicating the values of a, b, and c respectively.

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

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

print(a, pargs)

>>> func(1, 2, 3)

1 (2, 3)

1. The function func is defined with two parameters: a and \*pargs. The \*pargs parameter is preceded by an asterisk (\*) which indicates that it is a variable-length argument tuple.
2. When the function is called as func(1, 2, 3), the value 1 is assigned to the parameter a, and the remaining arguments 2 and 3 are collected into the pargs tuple.
3. The print statement inside the function will output 1 (2, 3), indicating the value of a and the tuple pargs containing the remaining arguments.
4. In this case, the \*pargs parameter allows the function to accept any number of additional positional arguments after the first argument a. The arguments after a are collected into a tuple, which can be accessed and processed within the function

4. What does this code print, and why?

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

print(a, kargs)

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

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

* The function func is defined with two parameters: a and \*\*kargs. The \*\*kargs parameter is preceded by two asterisks (\*\*) which indicates that it is a variable-length keyword argument dictionary.
* When the function is called as func(a=1, c=3, b=2), the keyword arguments c=3 and b=2 are collected into the kargs dictionary. The key-value pairs in the dictionary represent the names and values of the keyword arguments.
* The print statement inside the function will output 1 {'c': 3, 'b': 2}, indicating the value of a and the dictionary kargs containing the keyword arguments.
* In this case, the \*\*kargs parameter allows the function to accept any number of additional keyword arguments after the first argument a. The arguments after a are collected into a dictionary, which can be accessed and processed within the function.

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))

1 5 6 5

1. The function func is defined with four parameters: a, b, c, and d. The c parameter has a default value of 8, and the d parameter has a default value of 5.
2. When the function is called as func(1, \*(5, 6)), the first argument 1 is assigned to the a parameter. The \*(5, 6) notation unpacks the tuple (5, 6), and its elements are assigned to the remaining parameters in order. So, 5 is assigned to b, and 6 is assigned to c.
3. Since no value is provided for the d parameter, it takes its default value of 5.
4. The print statement inside the function will output 1 5 6 5, indicating the values of a, b, c, and d.
5. In this case, the \*(5, 6) syntax allows passing multiple values as arguments by unpacking them from a tuple. This can be useful when you have a sequence of values that you want to pass as individual arguments to a function.

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

(1, ['x'], {'a': 'y'})

* Initially, l is assigned the value 1, m is assigned a list [1], and n is assigned a dictionary {'a': 0}.
* When the function func is called with arguments l, m, and n, the following happens:
* The parameter a receives the value of l, which is 1.
* The parameter b receives the reference to the list object m.
* The parameter c receives the reference to the dictionary object n.
* Inside the function, the assignment a = 2 doesn't affect the original l variable because integers are immutable, and a is a new local variable.
* The assignment b[0] = 'x' modifies the list object m by changing its first element to 'x'.
* The assignment c['a'] = 'y' modifies the dictionary object n by adding a new key-value pair 'a': 'y'.
* After calling the function, the values of l, m, and n are printed, resulting in (1, ['x'], {'a': 'y'}).
* Notice that the list m and the dictionary n were mutable objects, so any modifications made inside the function affected the original objects.