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

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

print(a, b, c)

>>> func(1, 2)

In this code, the function func takes three arguments, a, b, and c, with default values for b and c. When you call func(1, 2), you pass values for a and b, but c retains its default value of 8. Therefore, the function prints the values of a, b, and c, which are 1, 2, and 8. The result is that it will print "1 2 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)

In this code, the function func takes three arguments, a, b, and c, with a default value for c. When you call func(1, c=3, b=2), you explicitly specify values for a, b, and c. The values of a, b, and c are 1, 2, and 3. The order of the arguments does not matter when you explicitly specify their names. The result is that it will print "1 2 3".

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

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

print(a, pargs)

>>> func(1, 2, 3)

This code defines the function func with a parameter a and \*pargs. The \*pargs syntax allows you to pass any number of positional arguments as a tuple. When you call func(1, 2, 3), a is assigned the value 1, and the rest of the positional arguments (2 and 3) are collected into the pargs tuple. Therefore, the function prints a (1) and pargs (a tuple containing (2, 3)). The result is that it will print "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)

In this code, the function func takes an argument a and \*\*kargs, which allows you to pass any number of keyword arguments as a dictionary. When you call func(a=1, c=3, b=2), you pass the values as keyword arguments. The value of a is 1, and the keyword arguments b and c are collected into the kargs dictionary. The function prints a (1) and kargs (a dictionary containing {'c': 3, 'b': 2}). The result is that it will print "1 {'c': 3, 'b': 2}".

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))

The function func takes three arguments, a, b, c, and d. When you call func(1, \*(5, 6)), the \* operator unpacks the tuple (5, 6) and assigns the values to the arguments in order. a gets 1, b gets 5, and c gets 6. There's no value provided for d, so it retains its default value of 5. The function prints the values of a, b, c, and d, which are 1, 5, 6, and 5. The result is that it will print "1 5 6 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

In this code, the function func takes three arguments, a, b, and c. Inside the function, a is assigned the value 2, b[0] is set to 'x', and c['a'] is set to 'y'. However, these changes are local to the function and do not affect the values of the variables l, m, and n defined outside the function. Therefore, when you print l, m, and n after calling func, they will still be 1, [1], and {'a': 0}, respectively. The result is that it will print "1", "[1]", and "{'a': 0}" for l, m, and n.