ASSIGNMENT - 23 (PYTHON)

1. What is the result of the code, and why?
>>> def func(a, b=6, c=8):
print(a, b, c)
>>> func(1, 2)

The code defines a function func with three parameters: a, b, and c, where b and c have default values of 6 and 8 respectively. When the function is called with func(1, 2), it provides values only for the first two parameters, a and b.

The output of the code will be:
1 2 8

Explanation:

a is assigned the value 1 because it's the first argument passed to the function.
b is assigned the value 2 because it's the second argument passed to the function.
c retains its default value of 8 because no argument is passed for it, so it keeps its default value.

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)

The code defines a function func with three parameters: a, b, and c, where c has a default value of 5. When the function is called with func(1, c=3, b=2), it provides values for all three parameters.

The output of the code will be:
1 2 3

Explanation:

a is assigned the value 1 because it's the first positional argument passed to the function.
b is explicitly assigned the value 2 because it's provided as a keyword argument.
c is explicitly assigned the value 3 because it's provided as a keyword argument, overriding its default value of 5.

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 a function func with a parameter a and *pargs, where *pargs collects any additional positional arguments into a tuple.

When the function func is called with func(1, 2, 3), it passes 1 as the value for a, and the additional positional arguments 2 and 3 are collected into the tuple pargs.

The output of the code will be: 1 (2, 3)

Explanation:

a is assigned the value 1 because it's the first positional argument passed to the function.
*pargs collects any additional positional arguments, in this case, 2 and 3, and stores them in a tuple. So pargs becomes (2, 3).

4. What does this code print, and why?
>>> def func(a, **kargs):
print(a, kargs)
>>> func(a=1, c=3, b=2)

This code defines a function func with a parameter a and **kargs, where **kargs collects any additional keyword arguments into a dictionary.

When the function func is called with func(a=1, c=3, b=2), it passes 1 as the value for a, and the additional keyword arguments c=3 and b=2 are collected into the dictionary kargs.

The output of the code will be: 1 {'c': 3, 'b': 2}

Explanation:

a is assigned the value 1 because it's provided explicitly as a keyword argument.
**kargs collects any additional keyword arguments, in this case, c=3 and b=2, and stores them in a dictionary. So kargs becomes {'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))

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

When func(1, *(5, 6)) is called, 1 is assigned to a, and the *(5, 6) syntax unpacks the tuple (5, 6) into positional arguments. So effectively, b gets 5 and c gets 6.

The output of the code will be: 1 5 6 5

Explanation:

a is assigned the value 1 because it's the first positional argument passed to the function.
b is assigned the value 5 because it's the first element of the tuple (5, 6) passed as positional arguments.
c is assigned the value 6 because it's the second element of the tuple (5, 6) passed as positional arguments.
d retains its default value of 5 because no value is passed for it.

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
```

The function func takes three parameters a, b, and c. Inside the function, a is reassigned to 2, the first element of list b is reassigned to 'x', and the value associated with key 'a' in dictionary c is reassigned to 'y'.

When func(l, m, n) is called with l=1, m=[1], and n={'a':0}, the following modifications occur:

a (which is a local variable inside the function) is reassigned to 2. This doesn't affect l because l is passed by value and a is a separate variable inside the function's scope.
The first element of list m is reassigned to 'x'. Since lists are mutable objects and m is passed by reference, this modification affects the original list m outside the function.
The value associated with key 'a' in dictionary n is reassigned to 'y'. Similar to the list m, dictionaries are mutable objects and n is passed by reference, so this modification affects the original dictionary n outside the function.
After the function call, the values of l, m, and n are:

l remains 1 because it's an integer, and the function only works with its local variable a.
m becomes ['x'] because the function modified the list m by changing its first element.
n becomes {'a': 'y'} because the function modified the value associated with the key 'a' in the dictionary n.
So, the result after the function call will be:
1, ['x'], {'a': 'y'}