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

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

print(a, b, c)

>>> func(1, 2)

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)

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

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

print(a, pargs)

>>> func(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)

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))

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

Answer:

1. The result of the code is:

```

1 2 8

```

In this function call, the argument `1` is assigned to the parameter `a`, and the argument `2` is assigned to the parameter `b`. Since no value is provided for the parameter `c`, it takes its default value of `8`. The `print` statement inside the function displays the values of `a`, `b`, and `c`, resulting in `1 2 8`.

2. The result of this code is:

```

1 2 3

```

In this function call, the argument `1` is assigned to the parameter `a`, the argument `2` is explicitly assigned to the parameter `b`, and the argument `3` is explicitly assigned to the parameter `c`. The `print` statement inside the function displays the values of `a`, `b`, and `c`, resulting in `1 2 3`. The order of the keyword arguments doesn't matter in this case.

3. The result of this code is:

```

1 (2, 3)

```

In this function call, the argument `1` is assigned to the parameter `a`, and the arguments `2` and `3` are collected into the `pargs` parameter as a tuple. The `print` statement inside the function displays the values of `a` and `pargs`, resulting in `1 (2, 3)`.

4. The code will print:

```

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

```

In this function call, the keyword arguments `a=1`, `c=3`, and `b=2` are collected into the `kargs` parameter as a dictionary. The `print` statement inside the function displays the values of `a` and `kargs`, resulting in `1 {'c': 3, 'b': 2}`.

5. The code will print:

```

1 5 6 5

```

In this function call, the argument `1` is assigned to the parameter `a`. The tuple `(5, 6)` is unpacked, and its elements `5` and `6` are respectively assigned to the parameters `b` and `c`. The parameter `d` takes its default value of `5`. The `print` statement inside the function displays the values of `a`, `b`, `c`, and `d`, resulting in `1 5 6 5`.

6. After executing the function call `func(l, m, n)`, the values of `l`, `m`, and `n` will be modified as follows:

```

l = 1

m = ['x']

n = {'a': 'y'}

```

Inside the `func` function, the statement `a = 2` reassigns the local variable `a` to `2`, but it doesn't affect the value of the variable `l` in the main program. The statement `b[0] = 'x'` modifies the first element of the list `m` to `'x'`, which is reflected in the variable `m` in the main program. The statement `c['a'] = 'y'` adds or modifies the key-value pair `'a': 'y'` in the dictionary `n`, which is also reflected in the variable `n` in the main program. Therefore, after the function call, `l` remains unchanged, `m` becomes `['x']`, and `n` becomes `{'a': 'y'}`.