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

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

print(a, b, c)

>>> func(1, 2)

ANS :- The result of the code will be:

```

1 2 8

```

Explanation:

In the function `func()`, there are 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`.

When we call the function `func(1, 2)`, we provide arguments for the parameters `a` and `b`, but not for `c`. In this case, the default value `8` for `c` is used.

Therefore, when the function is executed, it will print `1` for `a`, `2` for `b`, and `8` for `c`.

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)

ANS :- The result of the code will be:

```

1 2 3

```

Explanation:

In the function `func()`, there are three parameters: `a`, `b`, and `c`. The parameter `c` has a default value of `5`.

When we call the function `func(1, c=3, b=2)`, we provide the arguments for `a`, `b`, and `c` explicitly, specifying the values `1`, `2`, and `3` respectively.

Therefore, when the function is executed, it will print `1` for `a`, `2` for `b`, and `3` for `c`, reflecting the values passed in the function call.

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

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

print(a, pargs)

>>> func(1, 2, 3)

ANS :- The result of the code will be:

```

1 (2, 3)

```

Explanation:

In the function `func()`, the parameter `a` is a regular parameter, while `\*pargs` is a variable-length parameter or \*args. It allows the function to accept any number of additional positional arguments.

When we call the function `func(1, 2, 3)`, the value `1` is assigned to parameter `a`, and the values `2` and `3` are collected into a tuple assigned to `pargs`.

Therefore, when the function is executed, it will print `1` for `a` and `(2, 3)` for `pargs`, showing that the additional positional arguments are packed into a tuple.

4. What does this code print, and why?

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

print(a, kargs)

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

ANS :- The code will print:

```

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

```

Explanation:

In the function `func()`, the parameter `a` is a regular parameter, while `\*\*kargs` is a variable-length keyword parameter or \*\*kwargs. It allows the function to accept any number of additional keyword arguments.

When we call the function `func(a=1, c=3, b=2)`, the value `1` is assigned to parameter `a`, and the keyword arguments `c=3` and `b=2` are collected into a dictionary assigned to `kargs`.

Therefore, when the function is executed, it will print `1` for `a` and `{'c': 3, 'b': 2}` for `kargs`, showing that the additional keyword arguments are packed into a dictionary.

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))

ANS :- 1. The code `func(1, \*(5, 6))` will print:

```

1 5 6 5

```

Explanation:

The function `func()` has four parameters: `a`, `b`, `c`, and `d`. In the function call `func(1, \*(5, 6))`, the value `1` is assigned to `a`, and the tuple `(5, 6)` is unpacked and assigned to `b` and `c` respectively. Since `c` does not have a default value, it takes the first value from the tuple, which is `5`. The default value of `d` remains unchanged, which is `5`. Therefore, when the function is executed, 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

ANS :- The code `func(l, m, n)` will not print anything, but it will modify the values of `l`, `m`, and `n`.

After calling the function, the values of `l`, `m`, and `n` will be:

```

l = 1

m = ['x']

n = {'a': 'y'}

```

Explanation:

In the function `func(a, b, c)`, `a` is assigned a new value of `2`, `b[0]` (which is the first element of the list `m`) is assigned the value `'x'`, and `c['a']` (which is the value associated with the key `'a'` in the dictionary `n`) is assigned the value `'y'`.

As a result, after calling the function `func(l, m, n)`, the values of `l`, `m`, and `n` are modified accordingly. `l` remains unchanged as it is a primitive value, but `m` is a mutable list, so its first element is replaced with `'x'`, and `n` is a mutable dictionary, so the value associated with the key `'a'` is replaced with `'y'`.