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

Answers

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 result of the code is:

1 2 8

Explanation:

- The function `func` takes three parameters: `a`, `b`, and `c`.

- When the function is called with `func(1, 2)`, the following happens:

- `a` is assigned the value `1`

- `b` is assigned the value `2` since it is provided as the second argument

- `c` is assigned the default value of `8` since it is not provided as an argument

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 result of the code is:

1 2 3

Explanation:

- The function `func` takes three parameters: `a`, `b`, and `c` with a default value of `5` for `c`.

- When the function is called with `func(1, c=3, b=2)`, the following happens:

- `a` is assigned the value `1`

- `b` is assigned the value `2` since it is provided as a keyword argument

- `c` is assigned the value `3` since it is provided as a keyword argument

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

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

print(a, pargs)

>>> func(1, 2, 3)

The result of the code is:

1 (2, 3)

Explanation:

- The function `func` takes two parameters: `a` and `\*pargs`. The `\*pargs` parameter allows the function to accept any number of additional positional arguments.

- When the function is called with `func(1, 2, 3)`, the following happens:

- `a` is assigned the value `1`

- The remaining arguments `2` and `3` are collected into a tuple and assigned to `pargs`

4. What does this code print, and why?

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

print(a, kargs)

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

The result of the code is:

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

Explanation:

- The function `func` takes two parameters: `a` and `\*\*kargs`. The `\*\*kargs` parameter allows the function to accept any number of additional keyword arguments.

- When the function is called with `func(a=1, c=3, b=2)`, the following happens:

- `a` is assigned the value `1`

- The remaining keyword arguments `c=3` and `b=2` are collected into a dictionary and assigned to `kargs`

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 result of the code is:

1 5 6 5

Explanation:

- The function `func` takes four parameters: `a`, `b`, `c` (with a default value of `8`), and `d` (with a default value of `5`).

- When the function is called with `func(1, \*(5, 6))`, the following happens:

- `a` is assigned the value `1`

- The values `5` and `6` are unpacked from the tuple `(5, 6)` and assigned to `b` and `c` respectively

- `d` takes the default value of `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

The result of the code is:

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

Explanation:

- The function `func` takes three parameters: `a`, `b`, and `c`.

- Inside the function:

- `a` is assigned the value `2`, but this does not affect the original `l` value outside the function.

- `b[0]` is assigned the value `'x'`, which modifies the original list `m` outside the function.

- `c['a']` is assigned the value `'y'`, which modifies the original dictionary `n` outside the function.

- After the function call, the values of `l`, `m`, and `n` are:

- `l` is still `1` because the modification inside the function did not affect the original value.

- `m` is now `['x']` because the function modified the first element of the list.

- `n` is now `{'a': 'y'}` because the function added a new key-value pair to the dictionary.