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:

The code defines a function `func(a, b=6, c=8)` with three parameters: `a`, `b`, and `c`. The parameters `b` and `c` have default values of `6` and `8`, respectively. When the function `func()` is called with the arguments `func(1, 2)`, it matches the values of `1` and `2` to the first two parameters, `a` and `b`, respectively.Since there is no value provided for the parameter `c`, it takes its default value of `8`.The `print(a, b, c)` statement inside the function is then executed, which prints the values of `a`, `b`, and `c`.Therefore, the output will be `1 2 8`, where `a` is `1`, `b` is `2`, and `c` is `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)

ANS: The result of the code will be:

1 2 3

Explanation:

The code defines a function `func(a, b, c=5)` with three parameters: `a`, `b`, and `c`. The parameter `c` has a default value of `5`.When the function `func()` is called with the arguments `func(1, c=3, b=2)`, it matches the value `1` to the parameter `a`, the value `2` to the parameter `b`, and the value `3` to the parameter `c`.Although the order of the arguments is different from the function definition, the function call uses keyword arguments to explicitly specify which value goes to which parameter. Keyword arguments allow us to pass arguments in any order as long as we mention the parameter name.Since the value of `c` is explicitly specified as `3`, it takes the value `3`, overriding its default value of `5`. The `print(a, b, c)` statement inside the function is then executed, which prints the values of `a`, `b`, and `c`.

Therefore, the output will be `1 2 3`, where `a` is `1`, `b` is `2`, and `c` is `3`.

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:

The code defines a function `func(a, \*pargs)` with two parameters: `a` and `\*pargs`. The parameter `\*pargs` is a special syntax in Python called "Arbitrary Arguments" or "Variable-Length Arguments." It allows the function to accept an arbitrary number of additional positional arguments after the first argument `a`. These additional arguments are collected into a tuple named `pargs`.When the function `func()` is called with the arguments `func(1, 2, 3)`, the value `1` is matched to the parameter `a`, and the values `2` and `3` are collected into the `pargs` tuple. Inside the function, the `print(a, pargs)` statement is executed, which prints the value of `a` and the tuple `pargs`. The function prints `1` as the value of `a` and `(2, 3)` as the value of `pargs`.

Therefore, the output will be `1 (2, 3)`, where `a` is `1`, and `pargs` is the tuple `(2, 3)` containing the additional positional arguments passed to the function.

4. What does this code print, and why?

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

print(a, kargs)

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

ANS: The result of the code will be:

1 2 3

Explanation:

The code defines a function `func(a, b, c=5)` with three parameters: `a`, `b`, and `c`. The parameter `c` has a default value of `5`. When the function `func()` is called with the arguments `func(1, c=3, b=2)`, it matches the value `1` to the parameter `a`, the value `2` to the parameter `b`, and the value `3` to the parameter `c`. Although the order of the arguments is different from the function definition, the function call uses keyword arguments to explicitly specify which value goes to which parameter. Keyword arguments allow us to pass arguments in any order as long as we mention the parameter name. Since the value of `c` is explicitly specified as `3`, it takes the value `3`, overriding its default value of `5`.The `print(a, b, c)` statement inside the function is then executed, which prints the values of `a`, `b`, and `c`.

Therefore, the output will be `1 2 3`, where `a` is `1`, `b` is `2`, and `c` is `3`.

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: The result of the code will be `func(1, \*(5, 6))`. Let's revisit the correct explanation:

Code: `func(1, \*(5, 6))`

Result: `1 5 6 5`

Explanation:

The function `func(a, b, c=8, d=5)` expects at least two positional arguments: `a` and `b`, and can optionally take two more arguments: `c` and `d` with default values of `8` and `5`, respectively.In this code, the `\*` operator is used to unpack the tuple `(5, 6)` into two separate positional arguments. However, the tuple unpacking doesn't directly map to the function parameters. Instead, it first assigns the first value `5` to the parameter `b`, and the second value `6` is ignored. The value `1` is assigned to the parameter `a`. Since the tuple unpacking only provides one value, the parameters `c` and `d` take their default values of `8` and `5`, respectively. Therefore, the output of the function will be `1 5 8 5`.

So, the correct result is:

1 5 8 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 result of this code will be:

1 ['x'] {'a': 'y'}

Explanation:

a). The code defines a function `func(a, b, c)` which takes three parameters `a`, `b`, and `c`.

b). In the main code, variables `l`, `m`, and `n` are defined with initial values `1`, `[1]`, and `{'a': 0}` respectively.

c). When the function `func(l, m, n)` is called, the values of `l`, `m`, and `n` are passed to the function as arguments.

d). Inside the function `func(a, b, c)`, the following operations take place:

- `a` is assigned the value `2`, but since `a` is a local variable, it does not affect the value of `l` outside the function.

- The value at index 0 of list `b` (which is `m`) is changed to `'x'`. Since `m` is a mutable list, this change is reflected outside the function, and `m` becomes `['x']`.

- The value associated with the key `'a'` in dictionary `c` (which is `n`) is changed to `'y'`. Since `n` is a mutable dictionary, this change is also reflected outside the function, and `n` becomes `{'a': 'y'}`.

e). After the function call, the variables `l`, `m`, and `n` are printed using `l, m, n`, and their updated values are shown, which are `1`, `['x']`, and `{'a': 'y'}` respectively.

Hence, the output will be:

1 ['x'] {'a': 'y'}