**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 output of the code will be `1 2 8`.

In the function definition, `a` is a required argument, while `b` and `c` are optional arguments with default values of `6` and `8` respectively.

When the function is called with `func(1, 2)`, the argument `1` is assigned to the parameter `a` and `2` is assigned to the parameter `b`. Since no value is passed for `c`, the default value of `8` is used. Hence, the output is `1 2 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)**

The output of the code will be "1 2 3". The function is called with three arguments, where a is 1, b is 2 and c is explicitly set to 3. Since c is given an explicit value, the default value of 5 is not used. Therefore, the function prints the values of a, b, and c as 1, 2 and 3, respectively.

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

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

**print(a, pargs)**

**>>> func(1, 2, 3)**

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

In this function, `a` is a required positional argument, and `\*pargs` is a variable-length argument list that can accept any number of additional positional arguments.

When the function is called with `func(1, 2, 3)`, the value `1` is assigned to `a`, and the values `2` and `3` are collected into a tuple and assigned to `pargs`. The function then prints `1` followed by the tuple `(2, 3)`.

**4. What does this code print, and why?**

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

**print(a, kargs)**

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

The code will print `1 {'c': 3, 'b': 2}`.

Here, `a=1` is passed as a keyword argument, and `c=3` and `b=2` are passed as keyword arguments through `\*\*kargs`. The double-asterisk syntax `\*\*` in the parameter list of the function definition collects any additional keyword arguments that are passed to the function, and stores them in a dictionary.

Therefore, the output shows the value of the first parameter `a`, followed by the dictionary containing the remaining keyword arguments.

**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 output of the code will be `1 5 6 5`. Here, `(5, 6)` is a tuple passed to the function `func` using the `\*` operator which unpacks the tuple and passes its individual elements as positional arguments to the function. Thus, the function is called as `func(1, 5, 6)`, and since `c` and `d` are not passed explicitly, their default values of 8 and 5 respectively are used.

**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 code defines a function named `func` that takes three arguments `a`, `b`, and `c`. Inside the function, the values of `a`, the first element of `b`, and the value associated with key `'a'` in the dictionary `c` are changed.

The variables `l`, `m`, and `n` are then defined and initialized to `1`, `[1]`, and `{'a':0}`, respectively. The function `func` is called with `l`, `m`, and `n` as arguments. Since `l` is an integer, which is an immutable type, `l` is not modified inside the function. However, `m` is a list, which is mutable, so the first element of `m` is changed from `1` to `'x'`. Similarly, `n` is a dictionary, which is also mutable, so the value associated with key `'a'` in `n` is changed from `0` to `'y'`.

After the function call, the values of `l`, `m`, and `n` are printed, which results in the output:

```

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

```

So the value of `l` is unchanged, but `m` and `n` have been modified by the function call.