**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 provided code defines a function called `func` with three parameters `a`, `b`, and `c`. The function simply prints the values of these three parameters when called. The interesting thing about this function is that the parameters `b` and `c` have default values assigned to them (6 and 8, respectively).

Let's break down the function call `func(1, 2)`:

1. `a` is assigned the value 1 (the first argument).

2. `b` is assigned the value 2 (the second argument) since it is provided in the function call.

3. `c` is not provided in the function call, so it takes its default value, which is 8.

Now, the function will print these values:

```

1 2 8

```

So, the result of the code will be:

```

1 2 8

```

This is because the function is called with two arguments, which override the default values for `b` and `c` while `a` retains its provided value.

**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)**

In this code, we have a function called `func` with three parameters `a`, `b`, and `c`. The function prints the values of these three parameters when called.

Now, let's analyze the function call `func(1, c=3, b=2)`:

1. `a` is assigned the value 1 (the first positional argument).

2. `c` is explicitly assigned the value 3 using a keyword argument `c=3`.

3. `b` is explicitly assigned the value 2 using a keyword argument `b=2`.

Keyword arguments allow us to specify the parameter name explicitly, so their order doesn't matter in the function call.

Now, the function will print these values:

```

1 2 3

```

So, the result of the code will be:

```

1 2 3

```

This is because we provided specific values for all three parameters `a`, `b`, and `c` in the function call, and these values override any default values defined in the function signature.

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

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

**... print(a, pargs)**

**...**

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

In Python, when you define a function with an asterisk (\*) before a parameter in the parameter list, it allows that parameter to accept a variable number of positional arguments (also known as "args") into a tuple. These arguments will be packed into the tuple and can be accessed within the function.

Let's analyze the code step by step:

1. The function `func(a, \*pargs)` is defined with two parameters: `a` and `\*pargs`. Here, `a` is a regular parameter, and `\*pargs` is preceded by an asterisk, making it a variable-length argument that accepts multiple positional arguments.

2. When you call `func(1, 2, 3)`, you are passing three arguments: `1`, `2`, and `3`.

Now, let's see what the result would be when running the code:

```

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

... print(a, pargs)

...

>>> func(1, 2, 3)

```

Output:

```

1 (2, 3)

```

Explanation:

- The value of the regular parameter `a` is `1`.

- The variable-length argument `pargs` collects the remaining positional arguments after `a`. In this case, `pargs` becomes a tuple containing `(2, 3)`.

So, the function `func` prints `1 (2, 3)` as the result. The number `1` is the value of parameter `a`, and `(2, 3)` is the tuple containing the remaining positional arguments packed by `\*pargs`.

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

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

**... print(a, kargs)**

**...**

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

In Python, when you define a function with two asterisks (\*\*) before a parameter in the parameter list, it allows that parameter to accept a variable number of keyword arguments (also known as "kwargs") into a dictionary. These keyword arguments will be packed into the dictionary and can be accessed within the function.

Let's analyze the code step by step:

1. The function `func(a, \*\*kargs)` is defined with two parameters: `a` and `\*\*kargs`. Here, `a` is a regular parameter, and `\*\*kargs` is preceded by two asterisks, making it a variable-length argument that accepts multiple keyword arguments.

2. When you call `func(a=1, c=3, b=2)`, you are passing three keyword arguments: `a=1`, `c=3`, and `b=2`.

Now, let's see what the result would be when running the code:

```python

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

... print(a, kargs)

...

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

```

Output:

```

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

```

Explanation:

- The value of the regular parameter `a` is `1`.

- The variable-length argument `kargs` collects the keyword arguments passed after `a`. In this case, `kargs` becomes a dictionary containing the keyword arguments and their corresponding values: `{'c': 3, 'b': 2}`.

So, the function `func` prints `1 {'c': 3, 'b': 2}` as the result. The number `1` is the value of parameter `a`, and `{'c': 3, 'b': 2}` is the dictionary containing the keyword arguments packed by `\*\*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))**

In this code snippet, the function `func(a, b, c=8, d=5)` is defined with four parameters: `a`, `b`, `c`, and `d`. The parameters `c` and `d` have default values set to `8` and `5`, respectively. The function prints the values of these parameters.

Let's analyze the code step by step:

1. The function `func` is defined with four parameters: `a`, `b`, `c`, and `d`. When you define a function with default parameter values, it allows you to call the function with fewer arguments, and the missing arguments will take on their default values.

2. When you call `func(1, \*(5, 6))`, you are passing three arguments: `1`, `5`, and `6`.

Now, let's see what the result would be when running the code:

```python

>>> def func(a, b, c=8, d=5):

... print(a, b, c, d)

...

>>> func(1, \*(5, 6))

```

Output:

```

1 5 6 5

```

Explanation:

- The first argument `1` is assigned to parameter `a`.

- The remaining arguments `(5, 6)` are unpacked using the asterisk (\*) operator and assigned to parameters `b` and `c`, respectively. So, `b` gets the value `5`, and `c` gets the value `6`.

- Since there is no value provided for parameter `d`, it takes its default value of `5`.

As a result, the function `func` prints `1 5 6 5` because `a` is `1`, `b` is `5`, `c` is `6`, and `d` is `5`.

1. **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**

In this code, you have a function `func(a, b, c)` that takes three arguments `a`, `b`, and `c`. The function modifies the values of these arguments within the function body.

Let's analyze the code step by step:

1. You initialize three variables: `l` is set to `1`, `m` is assigned to a list containing `1`, and `n` is assigned to a dictionary with a single key-value pair, `'a': 0`.

2. When you call `func(l, m, n)`, you are passing the variables `l`, `m`, and `n` as arguments to the function.

3. Inside the function `func`, you perform the following actions:

- `a = 2;`: This sets the local variable `a` within the function to `2`. Note that this does not affect the original variable `l` outside the function because integers are immutable in Python.

- `b[0] = 'x';`: This modifies the first element of the list `m` passed as an argument to the function. After this line, the list `m` will be modified to `['x']`.

- `c['a'] = 'y';`: This modifies the value associated with the key `'a'` in the dictionary `n` passed as an argument to the function. After this line, the dictionary `n` will be modified to `{'a': 'y'}`.

Now, let's see what the result would be when printing the values of `l`, `m`, and `n` after calling the function:

```python

>>> 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

```

Output:

```

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

```

Explanation:

- The value of `l` remains unchanged, as it is an integer (immutable type) and the modifications to `a` within the function do not affect it.

- The list `m` is modified within the function using `b[0] = 'x'`, so when you print `m`, you get `['x']`.

- The dictionary `n` is also modified within the function using `c['a'] = 'y'`, so when you print `n`, you get `{'a': 'y'}`.

In summary, the result is `(1, ['x'], {'a': 'y'})`.