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

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

print(a, b, c)

>>> func(1, 2)

Sol.

1 2 8

The output here is '1 2 8', because 1 and 2 are passed to a and b by position, and c is omitted in the call and defaults to 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)

>>>sol. 1 2 3

The output this time is '1 2 3': 1 is passed to a by position, and b and c are passed 2 and 3 by name

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

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

print(a, pargs)

>>> func(1, 2, 3)

Sol. 1 (2, 3)

because 1 is passed to a and the \*pargs collects the remaining positional arguments into a new tuple object.

4. What does this code print, and why?

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

print(a, kargs)

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

Sol.

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

because 1 is passed to a by name and the \*\*kargs collects the remaining keyword arguments 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))

Sol.

1 5 6 5

1 matches a by position, 5 and 6 match b and c by \*name positionals (6 overrides c’s default), and d defaults to 5 because it was not passed a value.

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

Functions can send back results with return statements, by changing passed-in mutable arguments, and by setting global variables. Globals are generally frowned upon (except for very special cases, like multithreaded programs) because they can make code more difficult to understand and use. return statements are usually best, but changing mutables is fine, if expected. Functions may also communicate with system devices such as files and sockets, but these are beyond our scope here