**Assignment – 23**

Q1. 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**

When the function **func()** is called with the arguments **1** and **2**, the value **1** is assigned to the parameter **a**, and the value **2** is assigned to the parameter **b**. Since no value is provided for the parameter **c**, it takes its default value of **8**.

Therefore, when the **print()** statement is executed within the function, it prints the values of **a**, **b**, and **c**, which are **1**, **2**, and **8**, respectively.

Q2. What is the result of the code, and why?

>>>> def func(a, b=6, c=5):

print(a, b, c)

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

Sol.

**1 2 3**

In the function **func()**, when it is called with the arguments **1**, **c=3**, and **b=2**, the value **1** is assigned to the parameter **a**, **2** is assigned to the parameter **b**, and **3** is assigned to the parameter **c**.

Q3. 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)

In the function **func()**, when it is called with the arguments **1**, **2**, and **3**, the value **1** is assigned to the parameter **a**, and the remaining arguments **2** and **3** are collected into the tuple **pargs** due to the presence of the **\*** operator in the function definition.

Therefore, when the **print()** statement is executed within the function, it prints the value of **a**, which is **1**, and the tuple **pargs**, which contains the remaining arguments **2** and **3**.

Q4. 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}**

In the function **func()**, when it is called with the keyword arguments **a=1**, **c=3**, and **b=2**, the value **1** is assigned to the parameter **a**, and the additional keyword arguments **c=3** and **b=2** are collected into the **kargs** dictionary due to the presence of **\*\*kargs** in the function definition.

Q5. 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**

In the function **func()**, when it is called with the arguments **1** and **\*(5, 6)**, the value **1** is assigned to the parameter **a**. The **\*(5, 6)** syntax unpacks the tuple **(5, 6)** into individual positional arguments, which are then assigned to the parameters **b** and **c** respectively. Since no value is provided for the parameter **d**, it takes its default value of **5**.

Q6. 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

Sol.

**l** will still be **1**. **m** will be **['x']**. **n** will be **{'a': 'y'}**.

In the **func()** function, the parameters **a**, **b**, and **c** are received. Inside the function, the value of **a** is reassigned to **2**, but since it is a local variable, it does not affect the value of the variable **l** outside the function.

The element at index **0** of the list **b** is modified to **'x'**, which means the value of **m** will change because lists are mutable objects. The list **m** outside the function will be modified accordingly.

The value associated with the key **'a'** in the dictionary **c** is modified to **'y'**. Dictionaries are also mutable objects, so the dictionary **n** outside the function will reflect this modification.

After calling **func(l, m, n)**, the values of **l**, **m**, and **n** are printed.

**l** will still be **1** because the modification inside the function does not affect the variable outside the function.

**m** will be **['x']** because the modification inside the function changes the list object itself, which is reflected in the variable **m** outside the function.

**n** will be **{'a': 'y'}** because the modification inside the function changes the dictionary object itself, which is reflected in the variable **n** outside the function.