ANNEXURE

5

# Functions as Objects

## A5.1 INTRODUCTION

Python language gives a special treatment to functions. It treats functions as objects. This means that you can manipulate a function as any other object. This feature makes Python strikingly different from other OOP languages like Java or C#.

**Example A5.1** Program to demonstrate that function is an object

```
def func():
    """The function prints HELLO WORLD on screen """
print("HELLO WORLD")
print("Func is an instance of Object : ",isinstance(func, object))
print("ID(Func) : ", id(func))
print("Functions Docstring :", func.__doc__) # prints docstring
print("Function Name : ", func.__name__)  # prints function name
```

**OUTPUT**

```
HELLO WORLD
Func is an instance of Object :  True
ID(Func) :  47177648
Functions Docstring : The function prints HELLO WORLD on screens
Function Name :  func
```

Note that in the above program, `isinstance()` function checks whether the function is an instance of object. We will study later in this book that all entities in Python are inherited from the base class object. The function `id()` goes a step ahead and returns the object's id.

Like other objects, Python also allows you to store multiple functions in a tuple and then pass every entry in the tuple as an argument to another function.

**Example A5.2** Program to demonstrate that functions can be used as an argument to a function and can be stored in a collection

```
def func1():
    pass
def func2():
```

```
    pass
def func3():
    pass
def func4(func):
   print("ID(Func) = ",id(func))   # prints object id of function
Funcs = (func1, func2, func3, func4)   # Funcs is a tuple of functions
for i in Funcs:
    print(i)
    func4(i)   # Functions as argument
```

**OUTPUT**

```
<function func1 at 0x02B4DC30>
ID(Func) =  45407280
<function func2 at 0x02B4DFB0>
ID(Func) =  45408176
<function func3 at 0x02B56030>
ID(Func) =  45441072
<function func4 at 0x02B56070>
ID(Func) =  45441136
```

CASE STUDY

3

# Tower of Hanoi

The tower of Hanoi is one of the main applications of recursion. It says, 'if you can solve n–1 cases, then you can easily solve the nth case'. Look at Figure CS3.1 which shows three rings mounted on pole A.

The problem is to move all these rings from pole A to pole C while maintaining the same order. The main issue is that the smaller disk must always come above the larger disk.

We will be doing this using a spare pole. In our case, A is the source pole, C is the destination pole, and B is the spare pole. To transfer all the three rings from A to C, we will first shift the upper two rings (n–1 rings) from the source pole to the spare pole. We move the first two rings from pole A to B as shown in Figure CS3.2.

Now that n–1 rings have been removed from pole A, the nth ring can be easily moved from the source pole (A) to the destination pole (C). Figure CS3.3 shows this step.

The final step is to move the n–1 rings from the spare pole (B) to the destination pole (C). This is shown in Figure CS3.4.

To summarize, the solution to our problem of moving n rings from A to C using B as spare can be given as:

**Base case:** if n = 1

Σ Move the ring from A to C using B as spare

**Recursive case:**

Σ Move n – 1 rings from A to B using C as spare

Σ Move the one ring left on A to C using B as spare

Σ Move n – 1 rings from B to C using A as spare.

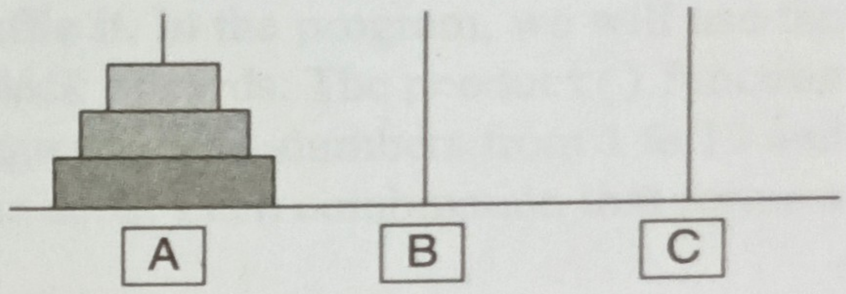

Figure CS3.1

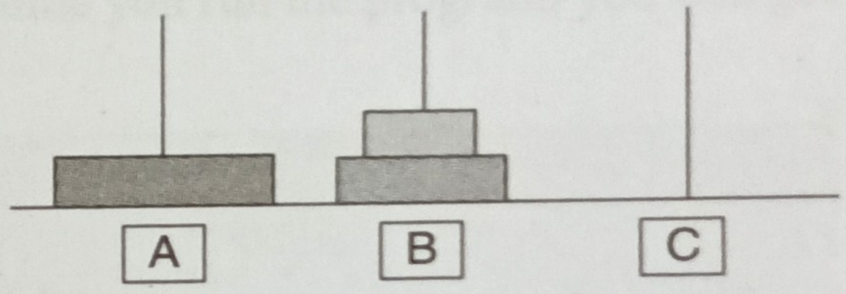

Figure CS3.2

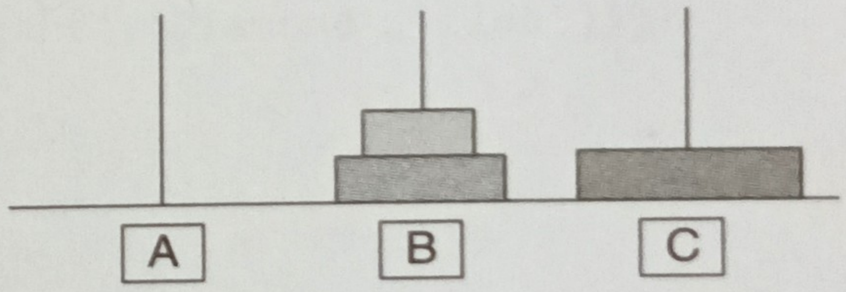

Figure CS3.3

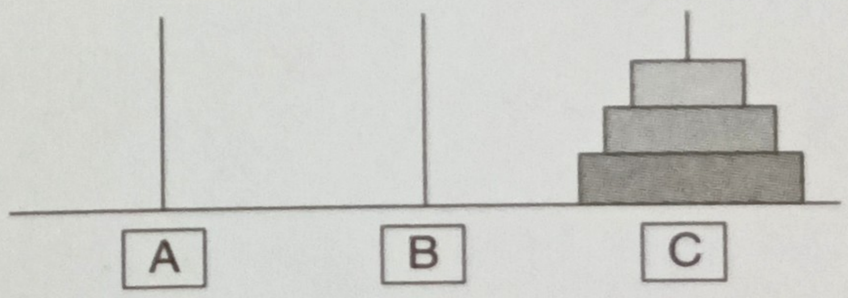

Figure CS3.4

The following code implements the solution of the tower of Hanoi problem.

```
# Program to implement tower of Hanoi

def hanoi(n, A, B, C):
    if n > 0:
        hanoi(n - 1, A, C, B)
        if A:
            C.append(A.pop())
        hanoi(n - 1, B, A, C)
```

```
A = [1,2,3,4]
C = []
B = []
hanoi(len(A),A,B,C)
print(A, B, C)
```

**OUTPUT**

```
[] [] [1, 2, 3, 4]
```

CASE STUDY

4

# Shuffling a Deck of Cards

Let us write a small code that will form a deck of cards and then shuffle it. In the program, we will use the `product()` function contained in the `itertools` module to create a deck of cards. The `product()` function performs the Cartesian product of the two sequence. Here, the two sequence are—numbers from 1 to 13 and the four suits. So, in all there are 13 * 4 = 52 combinations to form the deck. Each combination that forms a card is stored as a tuple. For example, `deck[0] = (1,'Spade')`.

Once the deck is formed, it is shuffled using the `shuffle()` function in the `random` module and then five cards are drawn and their combination is displayed to the user. Every time you run the program, you will get a different output.

```
# Program to shuffle a deck of cards
import itertools, random
# Form a deck of cards
deck = list(itertools.product(range(1,14),['Spade','Heart','Diamond','Club']))
# Shuffle the cards
random.shuffle(deck)
# Draw five cards
print("Your combination of cards is :")
for i in range(5):
   print(deck[i][0], "of", deck[i][1])
```

**OUTPUT**

```
Your combination of cards is :
11 of Heart
2 of Spade
13 of Club
7 of Club
4 of Club
```