CHAPTER 5

Functions and Modules

KEY Concepts

• Defining, Redefining, and Calling Functions • Variable Scope and Lifetime • return Statement • Required, Keyword, Default, and Variable Arguments • Lambda and Recursive Functions • Documentation Strings, Modules, and Packages • Standard Library, globals(), locals(), and reload()

5.1 INTRODUCTION

A *function* is a block of organized and reusable program code that performs a single, specific, and well-defined task. Python enables its programmers to break up a program into functions, each of which can be written more or less independently of the others. Therefore, the code of one function is completely insulated from the codes of the other functions.

Every function interfaces to the outside world in terms of how information is transferred to it and how results generated by the function are transmitted back from it. This interface is basically specified by the function name. For example, we have been using functions such as `input()` to take input from the user, `print()` to display some information on the screen, and `int()` to convert the user entered information into `int` datatype.

Let us consider Figure 5.1 which explains how a function `func1()` is called to perform a well-defined task. As soon as `func1()` is called, the program control is passed to the first statement in the function. All the statements in the function are executed and then the program control is passed to the statement following the one that called the function.

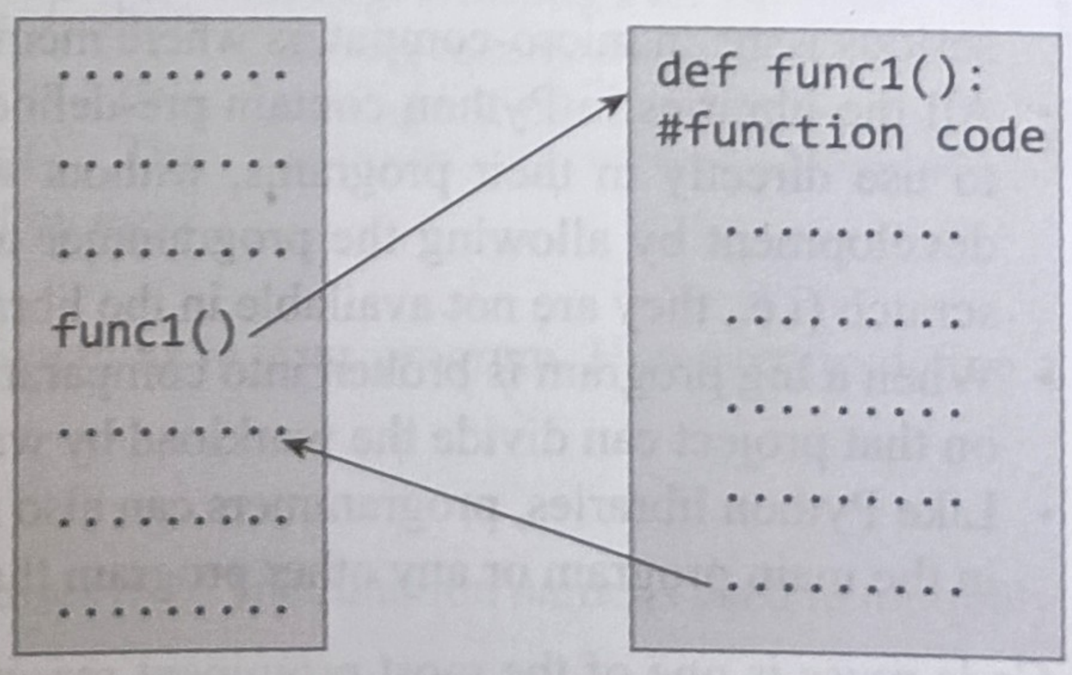

Figure 5.1 Calling a function

In the Figure 5.2 we see that `func1()` calls function named `func2()`. Therefore, `func1()` is known as the *calling function* and `func2()` is known as the *called function*. The moment the compiler encounters a function call, instead of executing the next statement in the calling function, the control jumps to the statements that are a part of the called function. After the called function is executed, the control is returned back to the calling program.

It is not necessary that the `func1()` can call only one function, it can call as many functions as it wants and as many times as it wants. For example, a function call placed within `for` loop or `while` loop may call the same function multiple times until the condition holds true.

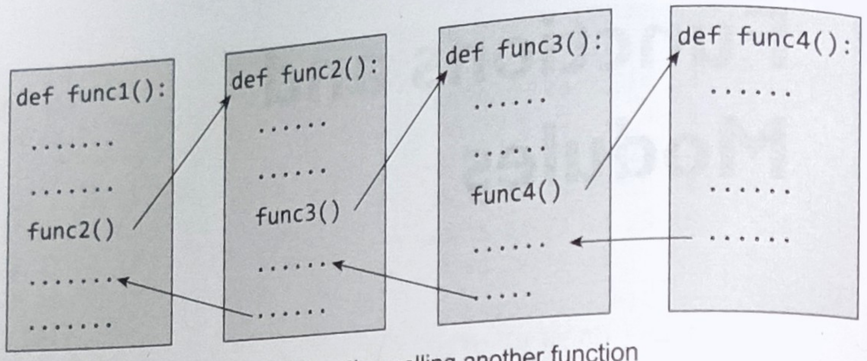

Figure 5.2 Function calling another function

Note Every function encapsulates a set of operations and when called it returns the information to the calling program.

5.1.1 Need for Functions

Let us analyze the reasons for segmenting a program into manageable chunks of code.

- Dividing the program into separate well defined functions facilitates each function to be written and tested separately. This simplifies the process of program development. Figure 5.3 shows that the *Function A* calls other functions for dividing the entire code into smaller sections (or functions).

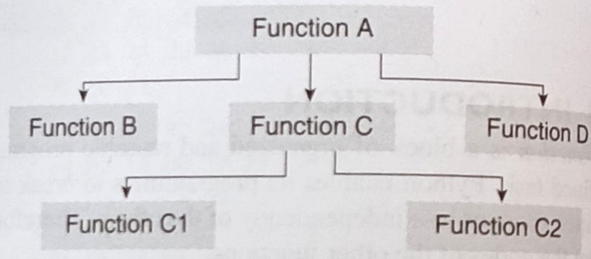

Figure 5.3 Top–down approach of solving a problem

- Understanding, coding, and testing multiple separate functions are far easier than doing the same for one huge function.
- If a big program has to be developed without the use of any function, then there will be a large number of lines in the code and maintaining that program will be a big mess. Also , the large size of the program is a serious issue in micro-computers where memory space is limited.
- All the libraries in Python contain pre-defined and pre-tested functions which the programmers are free to use directly in their programs, without worrying about their code details. This speeds up program development by allowing the programmer to concentrate only on the code that has to be written from scratch (i.e., they are not available in the libraries).
- When a big program is broken into comparatively smaller functions, then different programmers working on that project can divide the workload by writing different functions.
- Like Python libraries, programmers can also make their own functions and use them from different points in the main program or any other program that needs its functionalities.

Code reuse is one of the most prominent reason to use functions. Large programs usually follow the **DRY** principle, that is, *Don't Repeat Yourself* principle. Once a function is written, it can be called multiple times within the same or by a different program wherever its functionality is required. Correspondingly, a bad repetitive code abides by the **WET** principle, i.e., *Write Everything Twice, or We Enjoy Typing*.

Consider a program that executes a set of instructions repeatedly n times, though not continuously. In such case, the instructions had to be repeated continuously for n times they can better be placed within a loop. But if these instructions have to be executed abruptly from anywhere within the program code, then instead

of writing these instructions everywhere they are required, a better way is to place these instructions in a function and call that function wherever required. Figure 5.4 explains this concept.

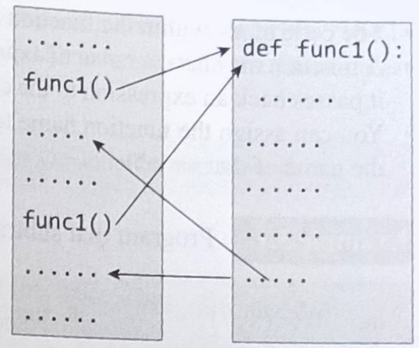

Figure 5.4 Function func1() called twice from the main module

Note Functions provide better modularity for your application and a high degree of code reuse.

5.2 FUNCTION DEFINITION

Any function can be compared to a black box (that is used for an entity having unknown implementation) that takes in input, processes it and then spits out the result. However, we may also have a function that does not take any inputs at all, or that does not return anything at all. While using functions we will be using the terminology given below.

- A function, *f* that uses another function *g*, is known as the *calling function* and *g* is known as the *called function*.
- The inputs that the function takes are known as *arguments/parameters*.
- When a called function returns some result back to the calling function, it is said to *return* that result.
- The calling function may or may not pass *parameters* to the called function. If the called function accepts arguments, the calling function will pass parameters, else not.
- *Function declaration* is a declaration statement that identifies a function with its name, a list of arguments that it accepts, and the type of data it returns.
- *Function definition* consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function.

Note Besides using built-in functions, users can also write their own functions. Such functions are called user defined functions.

Python gives functions first class treatment and gives them equal status with other objects. There are two basic types of functions, built-in functions and user defined ones. The built-in functions comes as a part of the Python language. For examples, `dir()`, `len()`, or `abs()`. The user defined functions, on the other hand, are functions created by users in their programs using the `def` keyword.

Programming Tip: Function naming follows the same rules of writing identifiers in Python.

As a Python programmer, you can write any number of functions in your program. However, to define a function, you must keep the following points in mind.

- Function blocks starts with the keyword `def`.
- The keyword is followed by the function name and parentheses (( )). The function name is used to uniquely identify the function.
- After the parentheses a colon (:) is placed.
- Parameters or arguments that the function accepts are placed within parentheses. Through these parameters values are passed to the function. They are optional. In case no values are to be passed, nothing is placed within the parenthesis.
- The first statement of a function can be an optional statement—the documentation string of the function or *docstring* describe what the function does. We will discuss this later in the book.

- The code block within the function is properly indented to form the block code.
- A function may have a `return[expression]` statement. That is, the return statement is optional. If it exits, it passes back an expression to the caller. A return statement with no arguments is the same as return None.
- You can assign the function name to a variable. Doing this will allow you to call the same function using the name of that variable.

Example 5.1 Program that subtracts two numbers using a function

```
def diff(x,y):          # function to subtract two numbers
        return x-y
a = 20
b = 10
operation = diff        # function name assigned to a variable
print(operation(a,b)) # function called using variable name
```

OUTPUT

```
10
```

Note The words before parentheses specifies the function name, and the comma-separated values inside the parentheses are function arguments.

When a function is defined, space is allocated for that function in the memory. A function definition comprises of two parts:

- Function header
- Function body

The syntax of a function definition can be given as:

```
def function_name(variable1, variable2,..)
      documentation string
      statement block
      return [expression]
```

Function Header

Function Body

Programming Tip: The indented statements form *body* of the function.

Example 5.2 To write a function that displays a string repeatedly

```
def func():
        for i in range(4):
                print("Hello World")
func()      #function call
```

OUTPUT

```
Hello World
Hello World
Hello World
Hello World
```

Programming Tip: The parameter list in the function definition as well as function declaration must match with each other.

In the aforementioned code, name of the function is `func`. It takes no arguments, and prints "`Hello World`" four times. The function is first defined before being called. The statements in the function are executed only when the function is called.

Note Before calling a function, you must define it just as you assign variables before using them.

5.3 FUNCTION CALL

Defining a function means specifying its name, parameters that are expected, and the set of instructions. Once the basic structure of a function is finalized, it can be executed by calling it.

The function call statement invokes the function. When a function is invoked, the program control jumps to the called function to execute the statements that are a part of that function. Once the called function is executed, the program control passes back to the calling function. The syntax of calling a function that does not accept parameters is simply the name of the function followed by parenthesis, which is given as,

Programming Tip: It is a logic error if the arguments in the function call are placed in a wrong order.

```
function_name()
```

Function call statement has the following syntax when it accepts parameters.

```
function_name(variable1, variable2, …)
```

When the function is called, the interpreter checks that the correct number and type of arguments are used in the function call. It also checks the type of the returned value (if it returns a value to the calling function).

Note List of variables used in function call is known as the actual parameter list. The actual parameter list may be variable names, expressions, or constants.

5.3.1 Function Parameters

A function can take parameters which are nothing but some values that are passed to it so that the function can manipulate them to produce the desired result. These parameters are normal variables with a small difference that the values of these variables are defined (initialized) when we call the function and are then passed to the function.

Parameters are specified within the pair of parentheses in the function definition and are separated by commas.

Key points to remember while calling the function

- The function name and the number of arguments in the function call must be same as that given in the function definition.
- If by mistake the parameters passed to a function are more than that it is specified to accept, then an error will be returned.

Example 5.3 Program to demonstrate the mismatch between function parameters and arguments

```
def func(i, j):
        print("Hello World", i, j)
```

```
func(5)
```

OUTPUT

```
TypeError: func() takes exactly 2 arguments (1 given)
```

- If by mistake the parameters passed to a function are less than that it is specified to accept, then an error will be returned.

Programming Tip: You can call a function from another function or directly from the Python prompt.

Example 5.4 Program to demonstrate the mismatch between function parameters and arguments

```
def func(i):
        print("Hello World", i)
func(5, 5)
```

OUTPUT

```
TypeError: func() takes exactly 1 argument (2 given)
```

- Names of variables in function call and header of function definition may vary.

Example 5.5 Program to demonstrate mismatch of name of function parameters and arguments

```
def func(i):                # function definition header accepts a variable with name i
    print("Hello World", i)
j = 10
func(j)                     # Function is called using variable j
```

OUTPUT

```
Hello World 10
```

- If the data type of the argument passed does not match with that expected in the function, then an error is generated.

Example 5.6 Program to demonstrate mismatch between data types of function parameters and arguments

```
def func(i):
   print("Hello World" + i)
func(5)
```

OUTPUT

```
TypeError: cannot concatenate 'str' and 'int' objects
```


- Arguments may be passed in the form of expressions to the called function. In such a case, arguments are first evaluated and converted to the type of formal parameter and then the body of the function gets executed.

Example 5.7 Program to demonstrate that the arguments may be passed in the form of expressions to the called function.

```
def func(i):
        print("Hello World", i)
func(5+2*3)
```

OUTPUT

```
Hello World 11
```

- The parameter list must be separated with commas.
- If the function returns a value then it may be assigned to some variable in the calling program. For example,

```
variable_name = function_name(variable1, variable2, ...);
```

Let us now try a program using a function.

Example 5.8 Program to add two integers using functions

```
def total(a,b):     # function accepting parameters
   result = a+b
   print("Sum of ", a, " and ", b, " = ", result)

a = int(input("Enter the first number : "))
b = int(input("Enter the second number : "))
total(a,b) #function call with two arguments
```

OUTPUT

```
Enter the first number : 10
Enter the second number : 20
Sum of 10 and 20 = 30
```

In the function `total()` used in the above program, we have declared a variable `result` just like any other variable. Variables declared within a function are called *local variables*. We will read more about it in the next section.

5.4 VARIABLE SCOPE AND LIFETIME

In Python, you cannot just access any variable from any part of your program. Some of the variables may not even exist for the entire duration of the program. In which part of the program you can access a variable and in which parts of the program a variable exists depends on how the variable has been declared. Therefore, we need to understand these two things:

- **Scope of the variable** Part of the program in which a variable is accessible is called its *scope*.
- **Lifetime of the variable** Duration for which the variable exists is called its *lifetime*.

5.4.1 Local and Global Variables

Global variables are those variables which are defined in the main body of the program file. They are visible throughout the program file. As a good programming habit, you must try to avoid the use of global variables because they may get altered by mistake and then result in erroneous output. But this does not mean that you should not use them at all. As a golden rule, use only those variables or objects that are meant to be used globally, like functions and classes, should be put in the global section of the program (i.e., above any other function or line of code).

Programming Tip: Trying to access local variable outside the function produces an error.

Correspondingly, a variable which is defined within a function is *local* to that function. A local variable can be accessed from the point of its definition until the end of the function in which it is defined. It exists as long as the function is executing. Function parameters behave like local variables in the function. Moreover, whenever we use the assignment operator (=) inside a function, a new local variable is created (provided a variable with the same name is not defined in the local scope).

Example 5.9 Program to understand the difference between local and global variables

```
num1 = 10    # global variable
print("Global variable num1 = ", num1)
def func(num2):          # num2 is function parameter
        print("In Function - Local Variable num2 = ",num2)
        num3 = 30        #num3 is a local variable
        print("In Function - Local Variable num3 = ",num3)
func(20)      #20 is passed as an argument to the function
print("num1 again = ", num1)       #global variable is being accessed
#Error- local variable can't be used outside the function in which it is defined
print("num3 outside function = ", num3)
```

OUTPUT

```
Global variable num1 = 10
In Function - Local Variable num2 = 20
In Function - Local Variable num3 = 30
num1 again =  10
num3 outside function =
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 12, in <module>
    print("num3 outside function = ", num3)
NameError: name 'num3' is not defined
```

Programming Tip: Variables can only be used after the point of their declaration

The following Table 5.1 lists the differences between global and local variables.

Table 5.1 Comparison Between Global and Local Variables

Global Variables	Local Variables
1. They are defined in the main body of the program file.	1. They are defined within a function and is *local* to that function.
2. They can be accessed throughout the program file.	2. They can be accessed from the point of its definition until the end of the block in which it is defined.
3. Global variables are accessible to all functions in the program.	3. They are not related in any way to other variables with the same names used outside the function.

5.4.2 Using the Global Statement

To define a variable defined inside a function as global, you must use the global statement. This declares the local or the inner variable of the function to have module scope. Look at the code given below and observe its output to understand this concept.

Example 5.10 Program to demonstrate the use of global statement

```
var = "Good"
def show():
        global var1
        var1 = "Morning"
        print("In Function var is - ", var)
show()
print("Outside function, var1 is - ", var1)      #accessible as it is global
variable
print("var is - ", var)
```

OUTPUT

```
In Function var is -  Good
Outside function, var1 is -  Morning
var is -  Good
```

Programming Tip: All variables have the scope of the block.

Key points to remember

- You can have a variable with the same name as that of a global variable in the program. In such a case a new local variable of that name is created which is different from the global variable. For example, look at the code in the following example and observe its output.

Example 5.11 Program to demonstrate name clash of local and global variable

```
var = "Good"
def show():
```

```
        var = "Morning"
        print("In Function var is - ", var)
show()
print("Outside function, var is - ", var)
```

OUTPUT

```
In Function var is -  Morning
Outside function, var is -  Good
```

- If we have a global variable and then create another global variable using the global statement, then changes made in the variable will be reflected everywhere in the program. This concept is illustrated in the code given below.

Example 5.12 Program to demonstrate modifying a global variable

```
var = "Good"
def show():
        global var
        var = "Morning"
        print("In Function var is - ", var)
show()
print("Outside function, var is - ", var)
var = "Fantastic"
print("Outside function, after modification, var is - ", var)
```

Programming Tip: You cannot assign value to a variable defined outside a function without using the global statement.

OUTPUT

```
In Function var is -  Morning
Outside function, var is -  Morning
Outside function, after modification, var is -  Fantastic
```

- In case of nested functions (function inside another function), the inner function can access variables defined in both outer as well as inner function, but the outer function can access variables defined only in the outer function. The following code explains this concept.

Programming Tip: Arguments are specified within parentheses. If there is more than one argument, then they are separated using comma.

Example 5.13 Program to demonstrate access of variables in inner and outer functions

```
def outer_func():
        outer_var = 10
        def inner_func():
                inner_var = 20
                print("Outer Variable = ", outer_var)
                print("Inner Variable = ", inner_var)
        inner_func()
```



```
   print("Outer Variable = ", outer_var)
   print("Inner Variable = ", inner_var)  #not accessible
outer_func()    #function call
```

OUTPUT

```
Outer Variable =  10
Inner Variable =  20
Outer Variable =  10
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 10, in <module>
  File ""C:\Python34\Try.py", line 9, in outer_func
NameError: name 'inner_var' is not defined
```

- If a variable in the inner function is defined with the same name as that of a variable defined in the outer function, then a new variable is created in the inner function. Look at the code given below to understand this concept.

Example 5.14 Program to demonstrate name clash variables in case of nested functions

```
def outer_func():
        var = 10
        def inner_func():
                var = 20
                print("Inner Variable = ", var)
        inner_func()
        print("Outer Variable = ", var)

outer_func()
```

OUTPUT

```
Inner Variable =  20
Outer Variable =  10
```

Note In the above program, even if we use global statement we would get the same result as global statement is applicable for the entire program and not just for outer function.

5.4.3 Resolution of Names

As discussed in the previous section, *scope* defines the visibility of a name within a block. If a local variable is defined in a block, its scope is that particular block. If it is defined in a function, then its scope is all blocks within that function.

When a variable name is used in a code block, it is resolved using the nearest enclosing scope. If no variable of that name is found, then a `NameError` is raised. In the code given below, `str` is a global string because it has been defined before calling the function.

Programming Tip: Try to avoid the use of global variables and global statement.

Example 5.15 Program that demonstrates using a variable defined in global namespace

```
def func():
        print(str)
str = "Hello World !!!"
func()
```

OUTPUT

```
Hello World !!!
```

You cannot define a local variable with the same name as that of global variable. If you want to do that, you must use the global statement. The code given below illustrates this concept.

Example 5.16 Program that demonstrates using a local variable with same name as that of global

```
def f():
        print(str) #global
        str = "Hello World!"  #local
        print(str)
str = "Welcome to Python Programming!"
f()
```

OUTPUT

```
UnboundLocalError: local variable 'str'
referenced before assignment
```

```
def f():
        global str
        print(str)
        str = "Hello World!"
        print(str)
str = "Welcome to Python Programming!"
f()
```

OUTPUT

```
Welcome to Python Programming!
Hello World!
```

5.5 THE return STATEMENT

In all our functions written above, no where we have used the `return` statement. But you will be surprised to know that every function has an implicit return statement as the last instruction in the function body. This implicit `return` statement returns nothing to its caller, so it is said to return `None`, where `None` means nothing. But you can change this default behavior by explicitly using the `return` statement to return some value back to the caller. The syntax of return statement is,

Programming Tip: A `return` statement with no arguments is the same as return `None`.

```
return [expression]
```

The expression is written in brackets because it is optional. If the expression is present, it is evaluated and the resultant value is returned to the calling function. However, if no expression is specified then the function will return `None`.

Note A function may or may not return a value.

The return statement is used for two things.

- Return a value to the caller
- To end and exit a function and go back to its caller

Example 5.17 Program to write a function without a `return` statement and try to print its return value. As mentioned earlier, such a function should return `None`.

```
def display(str):
   print(str)
x = display("Hello World") #assigning return value to another variable
print(x)
#print return value without assigning it to another variable
print(display("Hello Again"))
```

OUTPUT

```
Hello World
None
Hello Again
None
```

It should be noted that in the output `None` is returned from the function. The return value may or may not be assigned to another variable in the caller.

Example 5.18 Program to write another function which returns an integer to the caller

```
def cube(x):
   return (x*x*x)
num = 10
result = cube(num)
print("Cube of ", num, " = ", result)
```

OUTPUT

```
Cube of 10 = 1000
```

Note The return statement cannot be used outside of a function definition.

Key points to remember

- The return statement must appear within the function.
- Once you return a value from a function, it immediately exits that function. Therefore, any code written after the return statement is never executed. The program given in the following example illustrates this concept.

Example 5.19 Program to demonstrate flow of control after the `return` statement

```
def display():
   print("In Function")
   print("About to execute return statement")
   return
   print("This line will never be displayed")
```

```
display()
print("Back to the caller")
```

OUTPUT

```
In Function
About to execute return statement
Back to the caller
```

5.6 MORE ON DEFINING FUNCTIONS

We have already discussed in the previous section, the technique to define and call a function. In this section, we will go a step forward and learn some more ways of defining a function. All these features makes Python a wonderful language. Some of these features include

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

5.6.1 Required Arguments

We have already been using this type of formal arguments. In the *required arguments*, the arguments are passed to a function in correct positional order. Also, the number of arguments in the function call should exactly match with the number of arguments specified in the function definition.

Look at three different versions of `display()` given below and observe the output. The function displays the string only when number and type of arguments in the function call matches with that specified in the function definition, otherwise a `TypeError` is returned.

```
def display():
   print("Hello")
display("Hi")
```

OUTPUT

```
TypeError: display() takes
no arguments (1 given)
```

```
def display(str):
   print(str)
display()
```

OUTPUT

```
TypeError: display() takes
exactly 1 argument (0
given)
```

```
def display(str):
   print(str)
str ="Hello"
display(str)
```

OUTPUT

```
Hello
```

5.6.2 Keyword Arguments

When we call a function with some values, the values are assigned to the arguments based on their position. Python also allows functions to be called using keyword arguments in which the order (or position) of the arguments can be changed. The values are not assigned to arguments according to their position but based on their name (or keyword).

Programming Tip: Having a required argument after keyword arguments will cause error.

Keyword arguments when used in function calls, helps the function to identify the arguments by the parameter name. This is especially beneficial in two cases.

- First, if you skip arguments.
- Second, if in the function call you change the order of parameters. That is, in any order different from that specified in the function definition.

In both the cases mentioned above, Python interpreter uses keywords provided in the function call to match the values with parameters.

Example 5.20 Program to demonstrate keyword arguments

```
def display(str, int_x, float_y):
   print("The string is : ",str)
   print("The integer value is : ", int_x)
   print("The floating point value is : ", float_y)
display(float_y = 56789.045, str = "Hello", int_x = 1234)
```

OUTPUT

```
The string is:  Hello
The integer value is:  1234
The floating point value is:  56789.045
```

Example 5.21 Consider another program for keyword arguments in which during function call we use assignment operator to assign values to function parameters using other variables (instead of values).

```
def display(name, age, salary):
    print("Name : ", name)
    print("Age : ", age)
    print("Salary : ", salary)
n = "Aadi"
a = 35
s = 123456
display(salary = s, name = n, age = a)
```

OUTPUT

```
Name :  Aadi
Age :  35
Salary :  123456
```

Key points to remember

- All the keyword arguments passed should match one of the arguments accepted by the function.
- The order of keyword arguments is not important.
- In no case an argument should receive a value more than once.

Note Keyword arguments makes the program code easier to read and understand.

5.6.3 Default Arguments

Python allows users to specify function arguments that can have default values. This means that a function can be called with fewer arguments than it is defined to have. That is, if the function accepts three parameters, but function call provides only two arguments, then the third parameter will be assigned the default (already specified) value.

The default value to an argument is provided by using the assignment operator (=). Users can specify a default value for one or more arguments.

Note A default argument assumes a default value if a value is not provided in the function call for that argument.

Example 5.22 Program that uses default arguments

```
def display(name, course = "BTech"):
    print("Name : " + name)
    print("Course : ", course)
display(course = "BCA", name = "Arav") # Keyword Arguments
display(name = "Reyansh")              # Default Argument for course
```

OUTPUT

```
Name : Arav
Course :  BCA
Name : Reyansh
Course :  BTech
```

In the above code, the parameter name does not have a default value and is therefore mandatory. That is, you must specify a value for this parameter during the function call. But, parameter, course has already been given a default value, so it is optional. If a value is provided, it will overwrite the default value and in case a value is not specified during function call, the one provided in the function definition as the default value will be used.

Key points to remember

- You can specify any number of default arguments in your function.
- If you have default arguments, then they must be written after the non-default arguments. This means that non-default arguments cannot follow default arguments. Therefore, the line of code given in the following example will produce an error.

Example 5.23 Program to demonstrate default arguments

```
def display(name, course = "BTech", marks):  #error
    print("Name : " + name)
    print("Course : ", course)
    print("Marks : ", marks)
display(name = "Reyansh", 90)
```

Programming Tip: All the arguments to the right of the default argument must also have default values.

OUTPUT

```
SyntaxError: non-default argument follows default argument
```

Note A positional argument is assigned based on its position in the argument list but a keyword argument is assigned based on parameter name.

5.6.4 Variable-length Arguments

In some situations, it is not known in advance how many arguments will be passed to a function. In such cases, Python allows programmers to make function calls with arbitrary (or any) number of arguments.

When we use arbitrary arguments or variable-length arguments, then the function definition uses an asterisk (*) before the parameter name. Syntax for a function using variable arguments can be given as,

```
def functionname([arg1, arg2,.... ] *var_args_tuple ):
   function statements
   return [expression]
```

Example 5.24 Program to demonstrate the use of variable-length arguments

```
def func(name, *fav_subjects):
    print("\n", name, " likes to read ")
    for subject in fav_subjects:
        print(subject)
func("Goransh", "Mathematics", "Android Programming")
func("Richa", "C", "Data Structures", "Design and Analysis of Algorithms")
func("Krish")
```

OUTPUT

```
Goransh  likes to read  Mathematics Android Programming
Richa likes to read  C Data Structures Design and Analysis of Algorithms
Krish  likes to read
```

In the above program, in the function definition, we have two parameters—one is name and the other is variable-length parameter fav_subjects. The function is called three times with 3, 4, and 1 parameter(s) respectively. The first value is assigned to name and the other values are assigned to parameter fav_subjects. Everyone can have any number of favorite subjects and some can even have none. So when the third call is made, fav_subjects has no value and hence the for loop will not execute as there is no subject available in fav_subjects.

Key points to remember

- The arbitrary number of arguments passed to the function basically forms a tuple (data structure discussed later in this book) before being passed into the function.
- Inside the called function, for loop is used to access the arguments.
- The variable-length arguments if present in the function definition should be the last in the list of formal parameters.
- Any formal parameters written after the variable-length arguments must be keyword-only arguments.

Note A function cannot be used on the right side of an assignment statement. Therefore writing, *total(a, b) = s;* is invalid.

5.7 LAMBDA FUNCTIONS OR ANONYMOUS FUNCTIONS

Lambda or anonymous functions are so called because they are not declared as other functions using the def keyword. Rather, they are created using the lambda keyword. Lambda functions are throw-away

functions, i.e. they are just needed where they have been created and can be used anywhere a function is required. The lambda feature was added to Python due to the demand from LISP programmers.

Note Lambda is simply the name of a letter 'L' in the Greek alphabet.

Lambda functions contain only a single line. Its syntax can be given as,

```
lambda arguments: expression
```

The arguments contain a comma separated list of arguments and the expression is an arithmetic expression that uses these arguments. The function can be assigned to a variable to give it a name.

Example 5.25 Program that adds two numbers using the syntax of `lambda` function

```
sum = lambda x, y: x + y
print("Sum = ", sum(3, 5))
```

OUTPUT

```
Sum = 8
```

In the above code, the `lambda` function returns the sum of its two arguments. In the above program, `lambda x, y: x + y` is the `lambda` function. `x` and `y` are the arguments, and `x + y` is the expression that gets evaluated and returned. Note that the `lambda` function has no name. It returns a function object which is assigned to the identifier sum. Moreover,

```
lambda x, y: x + y
```

is same as writing,

```
def sum(x,y):
    return x+y
```

Note You can use `lambda` functions wherever function objects are required.

Key points to remember

Programming Tip: Lambda functions are not equivalent to inline functions in C/ C++.

- Lambda functions have no name.
- Lambda functions can take any number of arguments.
- Lambda functions can return just one value in the form of an expression.
- Lambda function definition does not have an explicit **return** statement but it always contains an expression which is returned.
- They are a one-line version of a function and hence cannot contain multiple expressions.
- They cannot access variables other than those in their parameter list.
- Lambda functions cannot even access global variables.
- You can pass lambda functions as arguments in other functions. Look at the code given in the following example to see how this is possible.

Example 5.26 Program to find smaller of two numbers using `lambda` function

```
def small(a,b):                    # a regular function that returns smaller value
    if(a<b):
        return a
    else:
        return b
sum = lambda x, y : x+y            # lambda function to add two numbers
diff = lambda x, y : x-y           # lambda function to subtract two numbers
#pass lambda functions as arguments to the regular function
print("Smaller of two numbers = ", small(sum(-3, -2), diff(-1, 2)))
```

Programming Tip: If you find lambda functions difficult, better use normal functions for clarity.

OUTPUT

```
Smaller of two numbers = -5
```

- Lambda functions are used along with built-in functions like `filter()`, `map()`, `reduce()`, etc. We will discuss these functions in later chapters.
- You can use lambda functions in regular functions.

Example 5.27 Program to use a `lambda` function with an ordinary function

```
def increment(y):
    return (lambda x: x+1)(y)
a = 100
print("a = ", a)
print("a after incrementing = ")
b = increment(a)
print(b)
```

OUTPUT

```
a =  100
a after incrementing =  101
```

In the aforementioned code, the regular function increment accepts a value in y. It then passes y to a lambda function. The lambda function increments its value and finally the regular function returns the incremented value to the caller.

- You can use a lambda function without assigning it to a variable. This is shown below.

```
# lambda function assigned to variable twice

twice = lambda x: x*2
print(twice(9))
```

OUTPUT

```
18
```

Argument passed to lambda function

```
# lambda function not assigned to any
variable twice

print ((lambda x: x*2) (9))
(twice(9))
```

OUTPUT

```
18
```

Argument passed to lambda function

You can pass lambda arguments to a function. This is shown in the code given below.

Example 5.28 Program that passes `lambda` function as an argument to a function

```
def func(f, n):
    print(f(n))

twice = lambda x: x * 2
thrice = lambda x: x * 3

func(twice, 4)
func(thrice, 3)
```

OUTPUT

```
8
9
```

- You can define a lambda that receives no arguments but simply returns an expression. Look at the code given in the following example.

Example 5.29 Program that uses a `lambda` function to find the sum of first 10 natural numbers

```
x = lambda: sum(range(1, 11))
# Invoke lambda expression that accepts no arguments but returns a value in y
print(x())
```

OUTPUT

```
55
```

Programming Tip: The print() returns None.

In the above code, we have assigned a variable x to a lambda expression and then invoked the lambda function with empty parentheses (without arguments).

- You can call a lambda function from another lambda unction. In such a case, the lambda function is said to be a nested function. However, use of nested lambda functions must be avoided. The program code given below demonstrates this concept.

Example 5.30 Program to add two numbers using `lambda` function

```
add = lambda x, y: x + y      #lambda function that adds two numbers
#lambda function that calls another lambda function to generate the result
multiply_and_add = lambda x, y, z: x * add(y,z)
print(multiply_and_add(3,4,5))
```

OUTPUT

```
27
```


Note With nested lambdas, recursion can occur and may also result in a RuntimeError: maximum recursion depth exceeded error.

- The time taken by a lambda function to perform a computation is almost similar to that take taken by a regular function.

5.8 DOCUMENTATION STRINGS

Docstrings (documentation strings) serve the same purpose as that of comments, as they are designed to explain code. However, they are more specific and have a proper syntax. As you can see below, they are created by putting a multiline string to explain the function. To understand the concept of documentation strings, let us first revisit the syntax of defining a function.

```
def functionname(parameters):
   "function_docstring"
   function statements
   return [expression]
```

As per the syntax, the first statement of the function body can optionally be a string literal which is also known as documentation string, or *docstring*. Docstrings are important as they help tools to automatically generate online or printed documentation. It also helps users and readers of the code to interactively browse through code. As a good programming habit, you must have a habit of including docstrings.

Key points to remember

- As the first line, it should always be short and concise highlighting the summary of the object's purpose.
- It should not specify information like the object's name or type.
- It should begin with a capital letter and end with a period.
- Triple quotes are used to extend the docstring to multiple lines. This docstring specified can be accessed through the __doc__ attribute of the function.
- In case of multiple lines in the documentation string, the second line should be blank, to separate the summary from the rest of the description. The other lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.
- The first non-blank line after the first line of the documentation string determines the amount of indentation for the entire documentation string.
- Unlike comments, docstrings are retained throughout the runtime of the program. So, users can inspect them during program execution.

Example 5.31 Program to show a multi-line docstring

```
def func():
    """The program just prints a message.
    It will display Hello World !!!   """
    print("Hello World !!!")
print(func.__doc__)
```

OUTPUT

```
Hello world!!!
The program just prints a message.
    It will display Hello World !!!
```

5.9 GOOD PROGRAMMING PRACTICES

While writing large and complex programs, you must take care of some points that will help you to develop readable, effective, and efficient code. For Python, **PEP 8** has emerged as the coding style guide that most projects adhere to promote a very readable and eye pleasing coding style. Some basic points that you should follow are:

- Instead of tabs, use 4 spaces for indentation.
- Insert blank lines to separate functions and classes, and statement blocks inside functions.
- Wherever required, use comments to explain the code.
- Use document strings that explains the purpose of the function.
- Use spaces around operators and after commas.
- Name of the classes should be written as `ClassName` (observe the capital letters, another example can be `StudentInfo`). We will read about classes in subsequent chapters.
- Name of the functions should be in lowercase with underscores to separate words. For example, `display_info()` and `get_data()`.
- Do not use non-ASCII characters in function names or any other identifier.

PROGRAMMING EXAMPLES

Program 5.1 Write a program using functions to check whether two numbers are equal or not.

```
def check_relation(a,b):
    if(a==b):
        return 0
    if(a>b):
        return 1
    if(a<b):
        return -1
a = 3
b = 5
res = check_relation(a,b)
if(res==0):
    print("a is equal to b")
if(res==1):
    print("a is greater than b")
if(res==-1):
    print("a is less than b")
```

Programming Tip: Function should be defined before it is called.

OUTPUT

```
a is less than b
```


Program 5.2 Write a program to swap two numbers.

```
def swap(a,b):
    a,b = b,a
    print("After swap : ")
    print("First number = ",a)
    print("Second number = ",b)
a = input("\n Enter the first number : ")
b = input("\n Enter the second number : ")
print("Before swap : ")
print("First number = ",a)
print("Second number = ",b)
swap(a,b)
```

OUTPUT

```
Enter the first number : 29
Enter the second number : 56
Before swap :
First number =  29
Second number =  56
After swap :
First number =  56
Second number =  29
```

Program 5.3 Write a program to return the full name of a person.

```
def name(firstName, lastName):
    separator = ' '
    n = firstName + separator + lastName
    return n
print(name('Janak', 'Raj'))
```

OUTPUT

```
Janak Raj
```

Program 5.4 Write a program to return the average of its arguments.

```
def avg(n1, n2):
    return (n1+n2)/2.0
n1 = int(input("Enter the first number : "))
n2 = int(input("Enter the second number : "))
print("AVERAGE = ", avg(n1,n2))
```

OUTPUT

```
Enter the first number : 5
Enter the second number : 7
AVERAGE =  6.0
```

Program 5.5 Write a program using functions and return statement to check whether a number is even or odd.

```
def evenodd(a):
      if(a%2==0):
            return 1
      else:
            return -1
a = int(input("Enter the number : "))
flag = evenodd(a)
if(flag==1):
      print("Number is even")
if(flag==-1):
      print("Number is odd")
```

OUTPUT

```
Enter the number : 1091
Number is odd
```

Program 5.6 Write a program to convert time into minutes.

```
def convert_time_in_min(hrs,minute):
      minute = hrs*60+minute
      return minute
h = int(input("Enter the hours : "))
m = int(input("Enter the minutes : "))
m = convert_time_in_min(h,m)
print("Minutes =",m)
```

OUTPUT

```
Enter the hours and minutes : 6
Enter the hours and minutes : 34
Minutes = 394
```

Program 5.7 Write a program to calculate simple interest. Suppose the customer is a senior citizen. He is being offered 12 per cent rate of interest; for all other customers, the ROI is 10 per cent.

```
def interest(p,y,s):
    if(s=='y'):
        SI = float((p*y*12)/100)
    else:
        SI = float((p*y*10)/100)
    return SI
p = float(input("Enter the principle amount : "))
y = float(input("Enter the number of years : "))
senior = input("Is customer senior citizen(y/n) : ")
print("Interest :",interest(p,y,senior))
```


OUTPUT

```
Enter the principle amount : 200000
Enter the number of years : 3
Is customer senior citizen(y/n) : n
Interest : 60000.0
```

Program 5.8 Write a program to calculate the volume of a cuboid using default arguments.

```
def volume(l,w=3,h=4):
      print("Length :",l,"\tWidth :",w,"\tHeight :",h)
      return l*w*h

print("Volume :",volume(4,6,2))
print("Volume :",volume(4,6))
print("Volume :",volume(4))
```

OUTPUT

```
Volume : Length : 4       Width : 6       Height : 2
48
Volume : Length : 4       Width : 6       Height : 4
96
Volume : Length : 4       Width : 3       Height : 4
48
```

Program 5.9 Write a program that computes P(n,r).

```
def fact(n):
    f = 1
    if(n==0 or n==1):
       return 1
    else:
      for i in range(1,int(n+1)):
            f = f*i
    return f

n = int(input("Enter the value of n : "))
r = int(input("Enter the value of r : "))
result = float(fact(n))/float(fact(r))
print("P(",str(n),"/",str(r),") = ",str(result))
```

OUTPUT

```
Enter the value of n : 9
Enter the value of r : 4
P( 9 / 4 ) =  15120.0
```

Program 5.10 Write a program to sum the series 1/1! + 4/2! + 27/3! + ...

```
def fact(n):
```

```
        f = 1
        if(n==0 or n==1):
              return 1
        else:
              for i in range(1,int(n+1)):
                    f = f*i
        return f

n = int(input("Enter the value of n : "))
s = 0.0
for i in range(1,n+1):
      s = s+(float(i**i)/fact(i))
print("Result :",s)
```

OUTPUT

```
Enter the value of n : 5
Result : 44.2083333333
```

Program 5.11 Write a program that uses docstings and variable-length arguments to add the values passed to the function.

```
def add(*args):
    '''Function returns the sum of values passed to it'''
    sum = 0
    for i in args:
       sum  += i
    return sum

print(add.__doc__)
print("SUM = ",add(25, 30, 45, 50))
```

OUTPUT

```
Function returns the sum of values passed to it
SUM =  150
```

Program 5.12 Write a program that greets a person.

```
def greet(name, mesg):
    """This function
    welcomes the person passed whose name
    is passed as a
    parameter"""

    print("Welcome, " + name + ". " + mesg)

mesg = "Happy Reading. Python is Fun !"
name = input("\n Enter your name : ")
greet(name, mesg)
```


OUTPUT

```
Enter your name : Goransh
Welcome, Goransh. Happy Reading. Python is Fun !
```

Program 5.13 Write a program to print the following pattern using default arguments.

```
%%%%%
^^^^^^
^^^^^^^^^^
^^^^^^^^^^^^^^
^^^^^^^^^^^^^^
def pattern(c='%',n=6,r=1):
     for i in range(r):
           print()
           for j in range(n):
                 print(c, end = ' ')

c = input("Enter the character to be displayed : ")
n = int(input("Enter the number of rows : "))
m = int(input("Enter the number of columns : "))
pattern()
pattern(c)
pattern(c,n)
pattern(c,n,m)
```

5.10 RECURSIVE FUNCTIONS

A *recursive function* is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Every recursive solution has two major cases, which are as follows:

- *base case*, in which the problem is simple enough to be solved directly without making any further calls to the same function.
- *recursive case,* in which first the problem at hand is divided into simpler sub-parts. Second, the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Thus, we see that recursion utilized divide and conquer technique of problem solving. *Divide and conquer technique* is a method of solving a given problem by dividing it into two or more smaller instances. Each of these smaller instances is recursively solved, and the solutions are combined to produce a solution for the original problem. Therefore, recursion is used for defining large and complex problems in terms of a smaller and more easily solvable problem. In a recursive function, a complicated problem is defined in terms of simpler problems and the simplest problem is given explicitly.

To understand recursive functions, let us take an example of calculating factorial of a number. To calculate `n!`, what we have to do is multiply the number with factorial of number that is 1 less than that number. In other words, `n! = n X (n-1)!`

Let us say we need to find the value of `5!...`

Programming Tip: Every recursive function must have at least one base case. Otherwise, the recursive function will generate an infinite sequence of calls thereby resulting in an error condition known as an infinite stack.

```
5! = 5 X 4 X 3 X 2 X 1
   = 120
```

This can be written as

`5! = 5 X 4!`, where
`4!= 4 X 3!`

Therefore,

`5! = 5 X 4 X 3!`

Similarly, we can also write,

`5! = 5 X 4 X 3 X 2!`

Expanding further

`5! = 5 X 4 X 3 X 2 X 1!`
We know, `1! = 1`

Therefore, the series of problem and solution can be given as shown in Figure 5.5.
Now if you look at the problem carefully, you can see that we can write a recursive function to calculate the factorial of a number. Note that we have said every recursive function must have a base case and a recursive case. For the factorial function,

- **Base case** is when n=1, because if `n = 1`, the result is known to be 1 as `1! = 1`.
- **Recursive case** of the factorial function will call itself but with a smaller value of n, this case can be given as

$$factorial(n) = n \times factorial\ (n\text{-}1)$$

PROBLEM	SOLUTION
5!	5 X 4 X 3 X 2 X 1!
= 5 X 4!	= 5 X 4 X 3 X 2 X 1
= 5 X 4 X 3!	= 5 X 4 X 3 X 2
= 5 X 4 X 3 X 2!	= 5 X 4 X 6
= 5 X 4 X 3 X 2 X 1!	= 5 X 24
	= 120

Figure 5.5 Recursive factorial function

Example 5.32 Program to calculate the factorial of a number recursively

```
def fact(n):
      if(n==1 or n==0):
            return 1
      else:
            return n*fact(n-1)
n = int(input("Enter the value of n : "))
print("The factorial of",n,"is",fact(n))
```

OUTPUT

```
Enter the value of n : 6
The factorial of 6 is 720
```

From the aforementioned example, let us analyze the basic steps of a recursive program.

Step 1: Specify the base case which will stop the function from making a call to itself.

Step 2: Check to see whether the current value being processed matches with the value of the base case. If yes, process and return the value.

Step 3: Divide the problem into a smaller or simpler sub-problem.

Step 4: Call the function on the sub-problem.

Step 5: Combine the results of the sub-problems.

Step 6: Return the result of the entire problem.

Note The base case of a recursive function acts as the terminating condition. So, in the absence of an explicitly defined base case, a recursive function would call itself indefinitely.

5.10.1 Greatest Common Divisor

The greatest common divisor of two numbers (integers) is the largest integer that divides both the numbers. We can find GCD of two numbers recursively by using the Euclid's algorithm that states:

$$GCD(a,b) = \begin{cases} b, & \text{if b divides a} \\ GCD(b,\ a \bmod b) & \text{otherwise} \end{cases}$$

`GCD()` can be implemented as a recursive function because if `b` does not divide `a`, then we call the same function (`GCD`) with another set of parameters that are smaller and simpler than the original ones. (Here we assume that `a > b`. However if `a < b`, then interchange `a` and `b` in the formula given above).

Working:
Assume `a = 62` and `b = 8`

```
GCD(62, 8)
      rem = 62 % 8 = 6
      GCD(8, 6)
            rem = 8 % 6 = 2
            GCD(6, 2)
                  rem = 6 % 2 = 0
            Return 2
      Return 2
Return 2
```

Program 5.14 Write a program to calculate GCD using recursive functions.

```
def GCD(x,y):
      rem = x%y
      if(rem==0):
            return y
      else:
            return GCD(y,rem)

n = int(input("Enter the first number : "))
m = int(input("Enter the second number : "))
print("The GCD of numbers is", GCD(n,m))
```

OUTPUT

```
Enter the first number :  50
```

```
Enter the second number :  5
The GCD of numbers is 5
```

5.10.2 Finding Exponents

We can find a solution to find exponent of a number using recursion. To find x^y, the base case would be when y=0, as we know that any number raise to the power 0 is 1. Therefore, the general formula to find x^y can be given as

$$EXP(x,y) = \begin{cases} 1, \text{ify} == 0 \\ x * EXP(x^{y-1}) \text{ otherwise} \end{cases}$$

Working:

```
exp_rec(2, 4) = 2 * exp_rec( 2, 3)
      exp_rec(2, 3) =  2 * exp_rec( 2, 2)
           exp_rec(2, 2) - 2 * exp_rec( 2, 1)
                exp_rec(2, 1)  = 2 * exp_rec( 2, 0)
                     exp_rec( 2, 0) = 1
                exp_rec( 2, 1) = 2 * 1 = 2
           exp_rec( 2, 2) = 2 * 2 = 4
     exp_rec( 2, 3) = 2 * 4 = 8
exp_rec( 2, 4) = 2 * 8 = 16
```

Program 5.15 Write a program to calculate exp(x,y) using recursive functions.

```
def exp_rec(x,y):
     if(y==0):
          return 1
     else:
          return (x*exp_rec(x,y-1))
n = int(input("Enter the first number : "))
m = int(input("Enter the second number : "))
print("Result = ", exp_rec(n,m))
```

OUTPUT

```
Enter the first number : 5
Enter the second number : 3
Result = 125
```

Note Recursive functions can become infinite if you don't specify the base case.

5.10.3 The Fibonacci Series

The Fibonacci series can be given as:

0 1 1 2 3 5 8 13 21 34 55.....

That is, the third term of the series is the sum of the first and second terms. On similar grounds, fourth term is the sum of second and third terms, so on and so forth. Now we will design a recursive solution to find the n^{th} term of the Fibonacci series. The general formula to do so can be given as

$$FIB(n) = \begin{cases} 1, \text{if } n<=2 \\ FIB(n-1) + FIB(n-2), \text{otherwise} \end{cases}$$

As per the formula, FIB(1) =1 and FIB(2) = 1. So we have two base cases. This is necessary because every problem is divided into two smaller problems. (Refer Figure 5.6)

Working:

If n = 7.

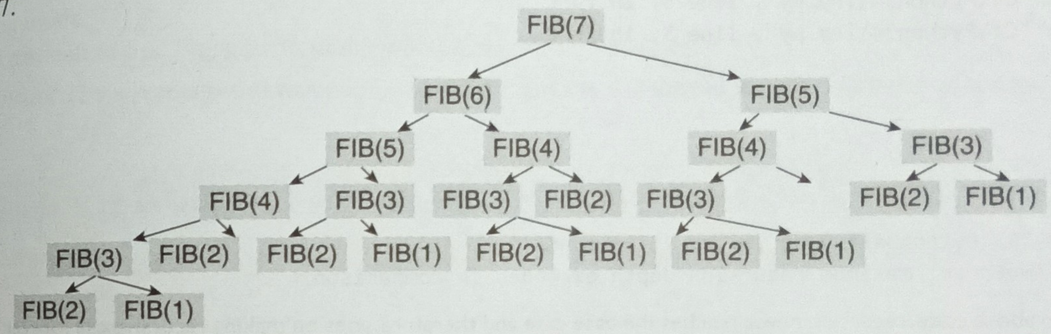

Figure 5.6: Recursion structure of FIB function

Note Recursion can also be indirect. That is, one function can call a second function which in turn calls the first, which again calls the second, and so on. This can occur with any number of functions.

Program 5.16 Write a program to print the Fibonacci series using recursion.

```
def fibonacci(n):
   if(n<2):
      return 1
   return (fibonacci(n-1)+fibonacci(n-2))

n = int(input("Enter the number of terms : "))
for i in range(n):
   print("Fibonacci(",i,") = ",fibonacci(i))
```

OUTPUT

```
Enter the number of terms : 5
Fibonacci( 0 ) =  1
Fibonacci( 1 ) =  1
Fibonacci( 2 ) =  2
Fibonacci( 3 ) =  3
Fibonacci( 4 ) =  5
```

Infinite Recursion

Let us consider the recursive program given below and observe the output.

```
def func(n, count=0):
    if n==0:
        return count
    else:
        return func(n, count+1)
print("Number of times recursive function was invoked = ", func(100))
```

OUTPUT

```
Traceback (most recent call last):
File "C:\Python34\Try.py", Line 6, in <module>
File "C:\Python34\Try.py", line 5, in func
File "C:\Python34\Try.py", line 5, in func
.
.
.
.
.
File "C:\Python34\Try.py", line 2, in func
RuntimeError: maximum recursion depth exceeded in comparison
```

In the above code, recursion never reaches the base case and therefore, goes on making recursive call forever. Such a recursive call is called ***infinite recursion***. To limit the side effects that can be caused by infinite recursion, Python reports a run-time error message when the maximum recursion depth is reached.

Recursion depth means the number of times a function is called. Python has specified maximum recursion depth to a value that is highly unlikely to be ever reached by any recursive function.

Note that usually Python allows not more than 1000 recursive calls thereby setting a limit in case of infinite recursion.

5.10.4 Recursion vs Iteration

Recursion is more of a top-down approach to problem solving in which the original problem is divided into smaller sub-problems. On the contrary, iteration follows a bottom-up approach that begins with what is known and then constructing the solution step-by-step.

Recursion is an excellent way of solving complex problems especially when the problem can be defined in recursive terms. For such problems a recursive code can be written and modified in a much simpler and clearer manner.

However, recursive solutions are not always the best solutions. In some cases recursive programs may require substantial amount of run-time overhead. Therefore, when implementing a recursive solution, there is a trade-off involved between the time spent in constructing and maintaining the program and the cost incurred in running time and memory space required for the execution of the program.

Whenever a recursive function is called, some amount of overhead in the form of a run-time stack is always involved. Before jumping to the function with a smaller parameter, the original parameters, the local variables, and the return address of the calling function are all stored on the system stack. Therefore, while using recursion a lot of time is needed to first push all the information on the stack when function is called and then time is again involved in retrieving the information stored on the stack once the control passes back to the calling function.

To conclude, one must use recursion only to find solution to a problem for which no obvious iterative solution is known. To summarize the concept of recursion, let us briefly discuss the pros and cons of recursion.

Pros The benefits of using a recursive program are:

- Recursive solutions often tend to be shorter and simpler than non-recursive ones.
- Code is clearer and easier to use.
- Recursion uses the original formula to solve a problem.
- It Follows a divide and conquer technique to solve problems.
- In some (limited) instances, recursion may be more efficient.

Cons The limitations of using a recursive program are:

- For some programmers and readers, recursion is a difficult concept.
- Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
- Aborting a recursive process in midstream is slow and sometimes nasty.
- Using a recursive function takes more memory and time to execute as compared to its non-recursive counterpart.
- It is difficult to find bugs, particularly when using global variables.

Conclusion The advantages of recursion pays off for the extra overhead involved in terms of time and space required.

Program 5.17 Write a program to count number of times a recursive function is called.

```
def func(n, count=0):
    if n==0:
        return count
    else:
        return func(n-1, count+1)
print("Number of times recursive function was invoked = ", func(100))
```

OUTPUT

```
Number of times recursive function was invoked =  100
```

Note Recursive functions make the code look clean and elegant.

5.11 MODULES

In the previous section, we have seen that functions help us to reuse a particular piece of code. Modules goes a step ahead. It allows you to reuse one or more functions in your programs, even in the programs in which those functions have not been defined.

Putting simply, module is a file with a `.py` extension that has definitions of all functions and variables that you would like to use even in other programs. The program in which you want to use functions or variables defined in the module will simply import that particular module (or `.py` file).

Note Modules are pre-written pieces of code that are used to perform common tasks like generating random numbers, performing mathematical operations, etc.

The basic way to use a module is to add import `module_name` as the first line of your program and then writing `module_name.var` to access functions and values with the name `var` in the module. Let us first use the standard library modules.

Example 5.33 Program to print the sys.path variable

```
import sys
print("\n PYTHONPATH = \n", sys.path)
```

OUTPUT

```
PYTHONPATH =
['C:\\Python34', 'C:\\Python34\\Lib\\idlelib', 'C:\\Windows\\system32\\python34.
zip', 'C:\\Python34\\DLLs', 'C:\\Python34\\lib', 'C:\\Python34', 'C:\\Python34\\
lib\\site-packages']
```

In the above code, we import the sys module (short form of system) using the import statement to use its functionality related to the Python interpreter and its environment.

When the `import sys` statement is executed, Python looks for the `sys.py` module in one of the directories listed in its `sys.path` variable. If the file is found, then the statements in the module is executed.

Module Loading and Execution

A module imported in a program must be located and loaded into memory before it can be used. Python first searches for the modules in the current working directory. If the module is not found there, it then looks for the module in the directories specified in the PYTHONPATH environment variable. If the module is still not found or if the PYTHONPATH variable is not defined, then a Python installation-specific path (like `C:\Python34\Lib`) is searched. If the module is not located even there, then an error `ImportError` exception is generated.

Till now, we have saved our modules in the same directory as that of the program importing it. But, if you want the module to be available to other programs as well, then the module should be either saved in the directory specified in the PYTHONPATH, or stored in the Python installation Lib directory.

Once a module is located, it is loaded in memory. A compiled version of the module with file extension `.pyc` is generated. Next time when the module is imported, this `.pyc` file is loaded, rather than the `.py` file, to save the time of recompiling. A new compiled version of a module is again produced whenever the compiled version is out of date (based on the dates when the `.pyc` file was created/modified). Even the programmer can force the Python shell to reload and recompile the `.py` file to generate a new `.pyc` file by using the `reload()` function.

5.11.1 The from...import statement

A module may contain definition for many variables and functions. When you import a module, you can use any variable or function defined in that module. But if you want to use only selected variables or functions, then you can use the `from...import` statement. For example, in the aforementioned program you are using only the path variable in the sys module, so you could have better written from sys import path.

Example 5.34 Program to show the use of `from...import` statement

```
from math import pi
print("PI = ", + pi)
```

OUTPUT

```
PI = 3.141592653589793
```


To import more than one item from a module, use a comma separated list. For example, to import the value of `pi` and `sqrt()` from the math module you can write,

```
from math import pi, sqrt
```

However, to import all the identifiers defined in the sys module, you can use the `fromsys import *` statement. However, you should avoid using the `import *` statement as it confuses variables in your code with variables in the external module.

Note This `imports *` statement imports all names except those beginning with an underscore (_).

You can also import a module with a different name using the **as** keyword. This is particularly more important when a module either has a long or confusing name.

Example 5.35 Program to show the use of `'as'` keyword

```
from math import sqrt as square_root
print(square_root(81))
```

OUTPUT

```
9.0
```

Python also allows you to pass command line arguments to your program. This can be done using the sys module. The `argv` variable in this module keeps a track of command line arguments passed to the `.py` script as shown below.

```
import sys
print(sys.argv)
```

To execute this program code, go to Command Prompt (in Windows) and write,

```
C:\Python34> python main.py Hello World
```

Thereafter, you will get the output as,

```
['main.py', 'Hello', 'World']
```

Program 5.18 Write a program to add two numbers that are given using command line arguments.

```
import sys
a = int(sys.argv[1])
b = int(sys.argv[2])
sum = a+b
print("SUM = ", sum)
```

OUTPUT

```
C:\Python34\python sum.py 3 4
SUM = 7
```

sys.exit() You can use `sys.exit([arg])` to exit from Python. Here, `arg` is an optional argument which can either be an integer giving the exit status or another type of object. If it is an integer, zero signifies successful termination and any non-zero value indicates an error or abnormal termination of the program. Most systems require the value of `arg` to be in the range `0-127`, and therefore produces undefined results otherwise. `None` is same as passing zero. If another type of object is passed, it results in an exit code of `1`. Generally, `sys.exit("Error Message")` is a quick way to exit a program when an error occurs.

Example 5.36 Program to demonstrate `sys.exit`

```
import sys
print("HELLO WORLD")
sys.exit(0)
```

OUTPUT

```
HELLO WORLD
```

5.11.2 Name of Module

Every module has a name. You can find the name of a module by using the `__name__` attribute of the module.

Example 5.37 Program to print the name of the module in which your statements is written

```
print("Hello")
print("Name of this module is : ", __name__)
```

OUTPUT

```
Hello
Name of this module is :  __main__
```

Observe the output and always remember that the for every standalone program written by the user the name of the module is `__main__`.

5.11.3 Making your own Modules

You can easily create as many modules as you want. In fact, you have already been doing that. Every Python program is a module, that is, every file that you save as `.py` extension is a module. The code given in the following example illustrates this concept.

First write these lines in a file and save the file as `MyModule.py`

```
def display():      #function definition
   print("Hello")
   print("Name of called module is : ", __name__)

str = "Welcome to the world of Python !!!     #variable definition
```

Then, open another file (`main.py`) and write the lines of code given as follows:


```
import MyModule
print("MyModule str = ", MyModule.str)      #using variable defined in MyModule
MyModule.display()                          #using function defined in MyModule
print("Name of calling module is : ", __name__)
```

When you run this code, you will get the following output.

```
MyModule str =  Welcome to the world of Python !!!
Hello
Name of called module is :  MyModule
Name of calling module is :  __main__
```

Note Modules should be placed in the same directory as that of the program in which it is imported. It can also be stored in one of the directories listed in sys.path.

Note that we have been using the dot operator to access members (variables or functions) of the module. Assuming that `MyModule` had many other variables or functions definition, we could have specifically imported just `str` and `display()` by writing the import statement as

```
from MyModule import str, display
```

Example 5.38 Program that defines a function large in a module which will be used to find larger of two values and called from code in another module

```
# Code in MyModule
def large(a,b):
   if a>b:
      return a
   else:
      return b

# Code in Find.py

import MyModule
print("Large(50, 100) = ", MyModule.large(50,100))
print("Large('B', 'c') = ", MyModule.large('B', 'c'))
print("Large('HI', 'BI') = ", MyModule.large('HI','BI'))
```

OUTPUT

```
Large(50, 100) =  100
Large('B', 'c') =  c
Large('HI', 'BI') =  HI
```

5.11.4 The dir() function

`dir()` is a built-in function that lists the identifiers defined in a module. These identifiers may include functions, classes, and variables. The `dir()` works as given in the following example.

Example 5.39 Program to demonstrate the use of `dir()` function

```
def print_var(x):
    print(x)
x = 10
print_var(x)
print(dir())
```

OUTPUT

```
10
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'print_var', 'x']
```

If you mention the module name in the `dir()` function, it will return the list of the names defined in that module. For example,

```
>>> dir(MyModule)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'display', 'str']
```

If no name is specified, the `dir()` will return the list of names defined in the current module. Just import the `sys` package and try to `dir` its contents. You will see a big list of identifiers. However, the `dir(sys)` does not list the names of built-in functions and variables. To see the list of those, write `dir(__builtin__)` as they are defined in the standard module `__builtin__`. This is shown below.

Example 5.40 Program to print all identifiers in the `dir()` function

```
import __builtin__
print(dir(__builtin__))
```

OUTPUT

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis',
'EnvironmentError', 'Exception', 'False', 'FloatingPointError', 'FutureWarning',
'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError',
'IndexError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError',
'NameError', 'None', 'NotImplemented', 'NotImplementedError', 'OSError',
'OverflowError', 'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True', 'TypeError',
'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',
'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
'WindowsError', 'ZeroDivisionError', '__debug__', '__doc__', '__import__',
'__name__', '__package__', 'abs', 'all', 'any', 'apply', 'basestring', 'bin',
'bool', 'buffer', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'cmp',
'coerce', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir',
'divmod', 'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float', 'format',
'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',
'int', 'intern', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list',
'locals', 'long', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct',
'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'raw_input', 'reduce',
'reload', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted',
'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'unichr', 'unicode',
'vars', 'xrange', 'zip']
```


Note By convention, modules are named using lowercase letters and optional underscore characters.

5.11.5 The Python Module

We have seen that a *Python module* is a file that contains some definitions and statements. When a Python file is executed directly, it is considered the *main module* of a program. Main modules are given the special name `__main__` and provide the basis for a complete Python program. The main module may *import* any number of other modules which may in turn import other modules. But the main module of a Python program cannot be imported into other modules.

5.11.6 Modules and Namespaces

A *namespace* is a container that provides a named context for identifiers. Two identifiers with the same name in the same scope will lead to a name clash. In simple terms, Python does not allow programmers to have two different identifiers with the same name. However, in some situations we need to have same name identifiers. To cater to such situations, namespaces is the keyword. Namespaces enable programs to avoid potential *name clashes* by associating each identifier with the namespace from which it originates.

```
# module1
def repeat_x(x):
    return x*2

# module2
def repeat_x(x):
    return x**2

import module1
import module2
result = repeat_x(10)     # ambiguous reference for identifier repeat_x
```

In the above example, `module1` and `module2` are imported into the same program. Each module has a function `repeat_x()`, which return very different results. When we call the `repeat_x()` from the main module, there will be a name clash as it will be difficult to determine which of these two functions should be called. Namespaces provide a means for resolving such problems.

In Python, each module has its own namespace. This namespace includes the names of all items (functions and variables) defined in the module. Therefore, two instances of `repeat_x()`, each defined in their own module, are distinguished by being fully qualified with the name of the module in which each is defined as, `module1.repeat_x` and `module2.repeat_x`. This is illustrated as follows:

```
import module1
import module2
result1 = module1.repeat_x(10)   # refers to repeat_x in module1
result2 = module2.repeat_x(10)   # refers to repeat_x in module2
```

Local, Global, and Built-in Namespaces

During a program's execution, there are three main namespaces that are referenced—the built-in namespace, the global namespace, and the local namespace. The *built-in namespace*, as the name suggests contains names of all the built-in functions, constants, etc. that are already defined in Python. The *global namespace* contains identifiers of the currently executing module and the *local namespace* has identifiers defined in the currently executing function (if any).

When the Python interpreter sees an identifier, it first searches the local namespace, then the global namespace, and finally the built-in namespace. Therefore, if two identifiers with the same name are defined in more than one of these namespaces, it becomes masked, as shown in the following example.

Example 5.41 Program to demonstrate name clashes in different namespaces

```
def max(numbers):        # global namespace
    print("USER DEFINED FUNCTION MAX.....")
    large = -1           # local namespace
    for i in numbers:
        if i>large:
            large = i
    return large
numbers = [9,-1,4,2,7]
print(max(numbers))
print("Sum of these numbers = ", sum(numbers))  #built-in namespace
```

OUTPUT

```
USER DEFINED FUNCTION MAX.....
9
Sum of these numbers =  21
```

In the aforementioned program, we have used function `max()` which is defined in the global namespace of the program. Local identifier, large is defined in a function. So it is accessible only in that function. Note that we have also used the function `sum()`. We have not given any definition of this function, so Python automatically uses the built-in version of the function.

Module Private Variables

In Python, all identifiers defined in a module are public by default. This means that all identifiers are accessible by any other module that imports it. But, if you want some variables or functions in a module to be privately used within the module, but not to be accessed from outside it, then you need to declare those identifiers as private.

In Python, identifiers whose name starts with two underscores (__) are known as private identifiers. These identifiers can be used only within the module. In no way, they can be accessed from outside the

module. Therefore, when the module is imported using the `import * form modulename`, all the identifiers of a module's namespace is imported except the private ones (ones beginning with double underscores). Thus, private identifiers become inaccessible from within the importing module.

Advantages of Modules Python modules provide all the benefits of modular software design. These modules provide services and functionality that can be reused in other programs. Even the standard library of Python contains a set of modules. It allows you to logically organize the code so that it becomes easier to understand and use.

Key points to remember

- A modules can import other modules.
- It is customary but not mandatory to place all import statements at the beginning of a module.
- A module is loaded only once, irrespective of the number of times it is imported.

5.12 PACKAGES IN PYTHON

A *package* is a hierarchical file directory structure that has modules and other packages within it. Like modules, you can very easily create packages in Python.

Remember that, every package in Python is a directory which must have a special file called `__init__.py`. This file may not even have a single line of code. It is simply added to indicate that this directory is not an ordinary directory and contains a Python package. In your programs, you can import a package in the same way as you import any module.

For example, to create a package called `MyPackage`, create a directory called `MyPackage` having the module `MyModule` and the `__init__.py` file. Now, to use `MyModule` in a program, you must first import it. This can be done in two ways.

```
import MyPackage.MyModule
```

or

```
from MyPackage import MyModule
```

The `__init__.py` is a very important file that also determines which modules the package exports as the `API`, while keeping other modules internal, by overriding the `__all__` variable as shown below.

```
__init__.py:
__all__ = ["MyModule"]
```

Key points to remember

- Packages are searched for in the path specified by `sys.path`.
- `__init__.py` file can be an empty file and may also be used to execute initialization code for the package or set the `__all__` variable.
- The import statement first checks if the item is defined in the package. If it is unable to find it, an `ImportError` exception is raised.
- When importing an item using syntax like import `item.subitem.subitem`, each item except the last must be a package. That is, the last item should either be a module or a package. In no case it can be a class or function or variable defined in the previous item.

- Packages have an attribute `__path__` which is initialized with a list having the name of the directory holding the `__init__.py` file. The `__path__` attribute can be modified to change the future searches for modules and sub-packages contained in the package.

Program 5.19 Write a program that prints absolute value, square root, and cube of a number.

```
import math

def cube(x):
    return x**3

a = -100
print("a = ", a)
a = abs(a)
print("abs(a) = ", a)
print("Square Root of ",a, " = ", math.sqrt(a))
print("Cube of ",a, " = ", cube(a))
```

OUTPUT

```
a =  -100
abs(a) =  100
Square Root of  100  =  10.0
Cube of  100  =  1000000
```

Programming Tip: * imports all objects from a module.

Program 5.20 Write a program to generate 10 random numbers between 1 to 100.

```
import random

for i in range(10):
        value = random.randint(1,100)
        print(value)
```

OUTPUT

```
66 68 14 7 76 8 70 43 60 70
```

5.13 STANDARD LIBRARY MODULES

Python supports three types of modules—those written by the programmer, those that are installed from external sources, and those that are pre-installed with Python. Modules that are pre-installed in Python are together known as the *standard library*. Some useful modules in the standard library are `string`, `re`, `datetime`, `math`, `random`, `os`, `multiprocessing`, `subprocess`, `socket`, `email`, `json`, `doctest`, `unittest`, `pdb`, `argparse`, and `sys`. You can use these modules for performing tasks like string parsing, data serialization, testing, debugging and manipulating dates, emails, command line arguments, etc.

Programming Tip: Most of the modules in Standard Library of Python are available on all platforms, but others are Windows or Unix specific.

Note Some of the modules in the Standard Library are written in Python, and others are written in C.

5.14 Globals(), Locals(), AND Reload()

The `globals()` and `locals()` functions are used to return the names in the global and local namespaces (In Python, each function, module, class, package, etc. owns a "**namespace**" in which variable names are identified and resolved). The result of these functions is of course, dependent on the location from where they are called. For example,

- If `locals()` is called from within a function, names that can be accessed locally from that function will be returned.
- If `globals()` is called from within a function, all the names that can be accessed globally from that function is returned.

Both the functions return names using dictionary. These names can be extracted using the `keys()` function. Dictionary data structure and `key()` will be discussed later in this book.

- `Reload()` When a module is imported into a program, the code in the module is executed only once. If you want to re-execute the top-level code in a module, you must use the `reload()` function. This function again imports a module that was previously imported. The syntax of the `reload()` function is gives as,

```
reload(module_name)
```

Here, `module_name` is the name of the module that has to be reloaded.

Program 5.21 Write a program to display the date and time using the `Time` module.

```
import time
localtime = time.asctime(time.localtime(time.time()) )
print("Local current time :", localtime)
```

OUTPUT

```
Local current time : Sun Dec 11 21:01:45 2016
```

Program 5.22 Write a program that prints the calendar of a particular month.

```
import calendar
print(calendar.month(2017, 1))
```

OUTPUT

```
    January 2017
Mo Tu We Th Fr Sa Su
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

Program 5.23 Write a program that uses the `getpass` module to prompt the user for a password, without echoing what they type to the console.

```
import getpass
password = getpass.getpass(prompt='Enter the password : ')
if password == 'oxford':
    print('Welcome to the world of Python Programming. ')
else:
    print('Incorrect password... Sorry, you cannot read our book.')
```

OUTPUT

```
Enter the password : oxford
Welcome to the world of Python Programming.
```

5.15 FUNCTION REDEFINITION

We have already learnt in the previous chapters that in Python, we can redefine a variable. That is you can change the value and even the type of value that the variable is holding. For example, in one line you can write `x = 5.6` and in the other line you can redefine `x` by writing `x = "Hello"`. Similar to redifining variables, you can also redefine functions in Python.

Programming Tip: Trying to import a module that is not available causes an `ImportError`.

Example 5.42 Program to demonstrate function redefinition

```
import datetime
def showMessage(msg):
    print(msg)
showMessage("Hello")
def showMessage(msg):
    now = datetime.datetime.now()
    print(msg)
    print(str(now))
showMessage("Current Date and Time is : ")
```

OUTPUT

```
Hello
Current Date and Time is :  2016-10-10 11:45:53.063000
```

In the above code, we have a function `showMessage()` which is first defined to simply display a message that is passed to it. After the function call, we have redefined the function to print the message as well as the current date and time.

Python Package Index (PyPI)

In Python, many third-party modules are stored in the **Python Package Index** (PyPI). To install them, you can use a program called *pip*. However, new versions of Python have these modules installed by default. Once you have these modules, installing libraries from PyPI becomes very easy. Simply, go to the command line (for Windows it will be the Command Prompt), and enter *pip install library_name*. Once the library is installed, import it in your program and use it in your code.

Using **pip** is the standard way of installing libraries on most operating systems, but some libraries have prebuilt binaries (executable files) for Windows which can be installed with a GUI the same way you would install other programs.