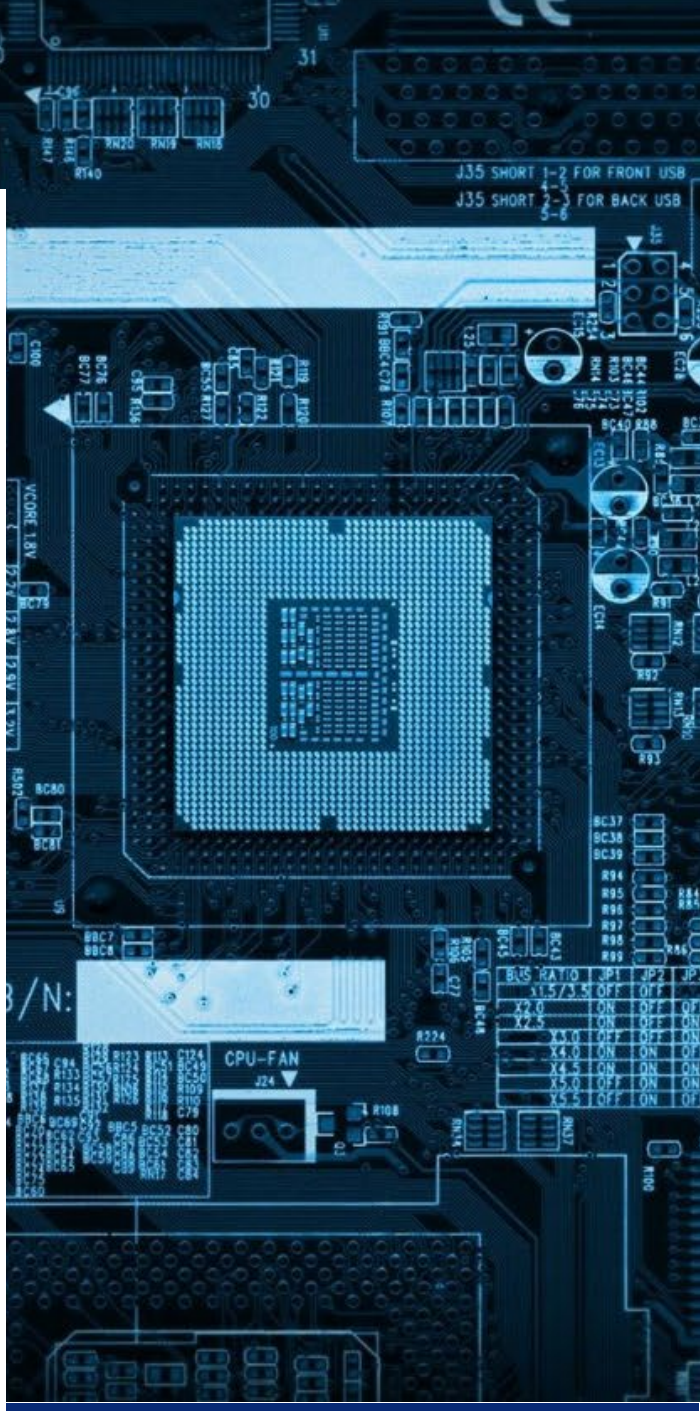
SCMA 249

Computer Programming in Actuarial Science

1/2022

LAB 6
Function



LAB 6
Function and Recursion

A function is a group of statements that exist within a program for the purpose of performing a specific task.

The advantages of using functions are:

- **Decomposing complex problems into simpler pieces**
  A program's code tends to be simpler and easier to understand when it is broken down into functions. Several small functions are much easier to read than one long sequence of statements.

- **Reuse of code**

  Would you rather write a single piece of code 10 times or just once and use it 10 times?

Source: edureka.co

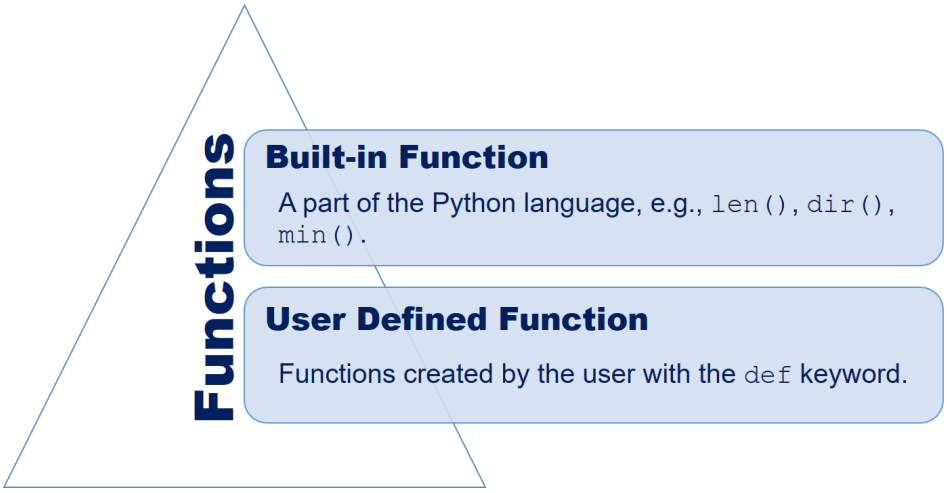
Function also reduces the duplication of code within a program. If a specific operation is performed in several places in a program, a function can be written once to perform that operation, then be executed any time is needed.

- **Reduce debugging time**
  When each task within a program is contained in its own function, testing and debugging becomes simpler. Programmers can test each function in a program individually, to determine whether it correctly performs its operation.

- **Faster development**
  Suppose a programmer or a team of programmers is developing multiple programs. They discover that each of the programs perform several common tasks, such as asking for a username and password, displaying the current time, and so on. It doesn't make sense to write the code for these tasks' multiple times.

Instead, functions can be written for the commonly needed tasks, and those functions can be incorporated into each program that needs them.

Like other language, there are 2 types of function in Python which are built-in and user-defined functions.



6.1 Defining and Calling Function

Similar to variable name, there are some rules for function name. Python requires the following rules:

- You cannot use one of Python's key words as a function name.
- A function name cannot contain spaces.
- The first character must be one of the letters `a` through `z`, `A` through `Z`, or an underscore character (_).
- After the first character you may use the letters `a` through `z`, `A` through `Z`, `0` through `9`, or underscores.
- Uppercase and lowercase characters are distinct.

Because functions perform actions, most programmers prefer to use verbs in function names.

To create a function, we use a syntax

```
def function_name(): #function header
    statement
    statement
```

The function header begins with the syntax `def`, followed by the function name, a set of parentheses, and a colon. The next line is a set of statements known as a block. A block is simply a set of statements that belong together as a group. These statements are performed any time the function is executed. Notice in the general format that all of the statements in the block are indented.

Sometimes it is useful not only to call a function, but also to send one or more pieces of data into the function. Pieces of data that are sent into a function are known as **arguments**.

The function can use its arguments in calculations or other operations. If you want a function to receive arguments when it is called, you must equip the function with one or more parameter variables. A **parameter variable**, often simply called a **parameter**, is a special variable that is assigned the value of an argument when a function is called.

Example 6.1 Defining and calling function with no argument

```
#Define a function named message
def message():
    print('This is an example of a function')
    print('which is not a complete program.')
```

This code defines a function named `message`. This function contains a block with two statements. To execute a function, you must call it.

```
#Call a function
message()
```

Example 6.2 Calling function with argument

```
def greeting(name):
    print('Hi,',name+'!')

#Main program-----------------------------------------
name = input('What is your name?\n')
greeting(name) #Call 'greeting' function
#-----------------------------------------------------
```

The variable `name` in the parentheses is called **`argument`**. The argument is optional input for a function. It is used when you want to pass some value to a function. A function can have several arguments.

When a function is called, the interpreter jumps to that function and executes the statement(s) in its block. Then, when the end of the block is reached, the interpreter jumps back to the part of the program that called the function, and the program resumes execution at that point.

Example 6.3 Returning value

The **`return`** statement is used to send the result of a function's calculations back to the caller. In this example, we use Python math module which contains trig functions worked in radians. We writhe a function to convert from radian to degree then calculate the sin value in degree.

```
from math import pi, sin
def deg_sin(x):
    return sin(pi*x/180)
```

The function **`deg_sin`** requires one argument, i.e., **`x`**, to execute. It returns value of $\sin(\pi x/180)$.

Example 6.4

Suppose we want to write a function that solves the system of equations $ax + by = e$ and $cx + dy = f$. It turns out that if there is a unique solution, then it is given by

$$x = \frac{de-bf}{ad-bc} \text{ and } y = \frac{af-ce}{ad-bc}.$$

We need our function to return both the x and y solutions.

```
def solve(a,b,c,d,e,f):
    x = (d*e-b*f)/(a*d-b*c)
    y = (a*f-c*e)/(a*d-b*c)
    return [x,y]

xsol, ysol = solve(2,3,4,1,2,5)
print('The solution is x = ', xsol, 'and y = ', ysol)
```

Example 6.5 Default arguments and keyword arguments

You can specify a default value for an argument. This makes it *optional*, and if the caller decides not to use it, then it takes the default value.

```
def multiple_print(string, n=1):
                            #n is an optional argument
    print(string * n,'\n')

multiple_print('สวัสดีค่ะ ',5)
multiple_print('สวัสดีค่ะ ')
```

6.2 Local and Global Variables

A **local variable** is created inside a function and cannot be accessed by statements that are outside the function. Different functions can have local variables with the same names because the functions cannot see each other's local variables.

Anytime you assign a value to a variable inside a function, you create a local variable. An error will occur if a statement in one function tries to access a local variable that belongs to another function.

Example 6.6

```
#Define the main function
def main():
    get_name()
    print('Hi,',name)

#Define the get_name function
def get_name():
    name = input('What is your name?\n')

main()
```

The output is

```
What is your name?
Kung
Traceback (most recent call last):
  File "C:/Users/SUNTAREE/Dropbox/Teaching/SCMA 249 Computer Programming in Actu
arial Science/Notes/LAB 8 Function and Recursion/Codes/EX-8-6.py", line 10, in <
module>
    main()
  File "C:/Users/SUNTAREE/Dropbox/Teaching/SCMA 249 Computer Programming in Actu
arial Science/Notes/LAB 8 Function and Recursion/Codes/EX-8-6.py", line 4, in ma
in
    print('Hi,',name)
NameError: name 'name' is not defined
```

This program has two functions: **main** and **get_name**. The **name** variable is entered by user in the **get_name** function. Hence, it is local to that function. It cannot be accessed by statements outside the **get_name** function.

On the other hand, sometimes you actually do want the same variable to be available to multiple functions. Such a variable is called a **global variable**. You have to be careful using global variables, especially in larger programs, but a few global variables used judiciously are fine in smaller programs.

Example 6.7

```
def sum_func(x,y):
    s = x + y
    return s

print(sum_func(8,6))
print(x)
```

Local variables can only be reached in their scope. Example 6.7 has two local variables: **x** and **y**. It will cause an error if you execute **print(x)** because **x** is a local variable in **sum_func** function.

Example 6.8

```
#This program calculates a retail item's sale price.

#DISCOUNT_PERCENTAGE is used as a global variable.
DISCOUNT_PERCENTAGE = 0.20

def main():
    #Get the item's regular price.
    reg_price = get_regular_price()

    #Calculate the sale price.
    sale_price = reg_price - discount(reg_price)

    #Display the sale price.
    print('The sale price is $',format(sale_price,',.2f'),sep='')

# The get_regular_price function prompts the use to enter
# an item's regular price and it returns that value.
def get_regular_price():
    price = float(input('Enter the item\'s regular price:'))
    return price

# The discount function accepts an item's price as an argument and returns
# the amount of the discount, specified by DISCOUNT_PERCENTAGE.
def discount(price):
    return price*DISCOUNT_PERCENTAGE

# Call the main function
main()
```

Example 6.9

```
def absolute_value(num):
        """This function returns the absolute
        value of the entered number"""

        if num >= 0:
                return num
        else:
                return -num

# Output: 2
print(absolute_value(2))

# Output: 4
print(absolute_value(-4))
```

Example 6.10

Create a function that writes the Fibonacci series to an arbitrary boundary.

```
def fib(n):
    'Print a Fibonacci series up to n.'
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

fib(10)
fib(500)
fib(1000)
```

Example 6.11 Python Function Unknown Number of Parameters

If the number of parameters a function should expect is unknown, then *args is added to the function definition as one of the parameters. This parameter expects a tuple. The asterisk (*) is important here. The name args is just a convention. It can be given any other name.

```
def calculate_sum(a, *args):
  sum = a
  for i in args:
    sum += i
  return sum

calculate_sum(10) # 10
calculate_sum(10, 11, 12) # 33
calculate_sum(1, 2, 94, 6, 2, 8, 9, 20, 43, 2) # 187
```

Example 6.12

```
def factorial(a):
    f=1
    for i in range(1,a+1):
        f=f*i
    return f

def combination(n,m):
    return factorial(n)/(factorial(m)*factorial(n-m))

print(combination(4,2))
```

References

- Kittipon P, Kittipob P, Somchai P, Sukree S. Python 101. V1.0.2., Chulalongkorn University Printing House; 2018.
- Andrew J. Python: The Ultimate Beginners Guide!., CreateSpace Independent Publishing Platform; 2016.
- Brian H. A Practical Introduction to Python Programming., Crative Commons Attribution-Noncommercial-Share Alike 3.0; 2015.
- https://www.thegeekstuff.com
- https://www.tutorialspoint.com