Chapter 5: Functions, Scope, and Namespaces

The creation and management of **functions** is critical to programming. Functions are collections of code that can accept inputs, perform tasks, and create outputs. To this point, this book has utilized very basic functions, including *main( )*. However, functions and their abilities, optional parameters, and return methods have not been defined or explored. This chapter will introduce functions, function parameters, and function return methods.

The syntactical form of a function is the following:

```
def <function name>(<function parameters>):
      <function statements>
```

5.1: Concepts for Functions

Functions are generally created to perform tasks that will likely be repeated. Functions can be called as many times as needed, can be nested into decision or loop structures, and can organize programs into more manageable collections of code. Python offers many built-in functions, such as print or input, which are used to accomplish tasks (without having to rewrite all the required steps each time).

For example, suppose there was no print function. Then, to print something as output, the programmer would have to code all the statements required to identify the output console, send information to that console, and display information on the console. And, each time a print was required, the same code would have to be rewritten or copied. This is time consuming and wasteful.

Functions not only offer reusability, but they also encapsulate code that is specific for a given task. All statements inside of a function belong to the **scope** of that function. This ownership protects the function from other activities performed in the program, and vice versa. The notion of functions can be extended to the idea of objects and classes (a topic of Chapter 10).

Functions offer program flow control. Most programs use the main function as the starting point of the program. Functions can call other functions, can be nested in functions, and can collect and return information.

As an option, the definition of a function can include **parameters** (sometimes called arguments). Parameters are often used to send information and values to a function. The **return** statement can be placed inside of a function, allowing the function to return a value, or other structure to the caller.

Example 5.1.1: A basic function definition

```
def NewFunction():
    print("This is a new function")
```

All functions must be defined. All statements within a function are indented (using the tab) to exist within the scope of the function definition. Recall that Python uses indentation to signify scope and ownership.

Example 5.1.2: Calling a basic function

```
NewFunction()
```

To call a function, type the name of the function, followed by parentheses. Because this function does not have any parameters, the parenthesis in the function call are empty. Similarly, because this function does not return a value, it is not necessary to set a variable equal to the function call.

Example 5.1.3: Defining a function with parameters

```
def NewFunction(name, age):
     print("Your name is ", name)
     print("Your age is ", age)
```

Example 5.1.4: Calling a function with parameters

```
NewFunction("Bob", 37)
```

When the function called, NewFunction, is called, it must be called with the number of parameters it was defined with. The NewFunction expects two parameters. Any two parameters can be used in this case. When `NewFunction("Bob", 37)` is called, the output will be:

```
Your name is  Bob
Your age is  37
```

Similarly, `NewFunction("Sally", 26)` will result in the output of,

```
Your name is  Sally
Your age is  26
```

Example 5.1.5: Returning a value from a function

Functions can return values, lists, and other structures using the **return** statement. The following will illustrate the syntax as well as example.

```
def CalculateCelsius(F):
    Celsius=(F - 32)*5/9
```

```
    return Celsius
```

Here, a function called CalculateCelsius is defined with one parameter. The parameter is named "F". The function uses the value sent as the parameter, F, to calculate the Celsius equivalent. The function then returns that value to the caller. The following small program illustrates three calls to this function. Notice that each call collects the return value.

```
def main():
    C1 = CalculateCelsius(30)
    C2 = CalculateCelsius(45)
    C3 = CalculateCelsius(60)

    print(C1, C2, C3)

def CalculateCelsius(F):
    Celsius=round((F - 32)*5/9,2)
    return Celsius

main()
```

The output from the above program is:

```
-1.11 7.22 15.56
```

Function parameters, such as "F" in the above example, can initially be confusing. The best way to think about function parameters is as "empty slots in memory." The "F" is a slot in memory that can hold any value (parameter) sent to the function.

When `CalculateCelsius(30)` is called, the value 30 is sent. When this happens, the "F" parameter in the function definition stores the value of 30 in its memory location. As such, "F" becomes 30 for all parts of the function. If F is 30, then `Celsius=round((F - 32)*5/9,2)` becomes `Celsius=round((30 - 32)*5/9,2),` which equals -1.11.

Similarly, when `CalculateCelsius(60)` is called, the value 60 is sent. When this happens, the "F" parameter in the function definition stores the value of 60 in its memory location. As such, "F" becomes 60 for all parts of the function. If F is 60, then `Celsius=round((F - 32)*5/9,2)` becomes `Celsius=round((60 - 32)*5/9,2),` which equals 15.56.

5.2: Using functions in programs

This section offers several program examples (including the output) for the use of functions in programs.

Example 5.2.1: Functions with Parameters

The following example program illustrates a program with three functions (including main). One of the three functions utilizes parameters. It is recommended that you type in each example program in this chapter, save and run the program, and then make alterations to the code to see what effects the alterations have.

```
     # FunctionExample1.py
     # by Ami Gates
     # This small program illustrates a function with parameters

     def main():
         num = eval(input("How many employees would you like to
enter? :"))
         while num > 0:
             getInfo()
             num=num-1


     def printInfo(name, ID, phone, age):
         print("Employee's full name :", name)
         print("Employee's ID:", ID)
         print("Employee's phone number:", phone)
         print("Employee's age:", age)


     def getInfo():
         n=input("Enter Employee's full name: ")
         id=input("Enter Employee's ID: ")
         ph=input("Enter Employee's phone number:  ")
         a=input("Enter Employee's age: ")
         printInfo(n,id,ph,a)


     main()
```

One possible input/output for this program is the following:

```
     How many employees would you like to enter? :2

     Enter Employee's full name: Anne Smith

     Enter Employee's ID: D3333
```

```
Enter Employee's phone number:  333-333-3333

Enter Employee's age: 33
Employee's full name : Anne Smith
Employee's ID: D3333
Employee's phone number: 333-333-3333
Employee's age: 33

Enter Employee's full name: John Bandy

Enter Employee's ID: D4545

Enter Employee's phone number:  444-555-4545

Enter Employee's age: 45
Employee's full name : John Bandy
Employee's ID: D4545
Employee's phone number: 444-555-4545
Employee's age: 45
```

This example program has several new elements. Starting at the first statement: The `main()` function is defined. However, the call to `main()` is actually the last line of code.

Functions must first be **defined,** and then they can be **called**. It is possible to define a function and never call it, but this would be unusual. If a function is not called, it will never execute. Alternatively, if a function is called on, but not defined, an error will occur and execution will discontinue because the interpreter will not know where to locate the function definition.

Functions are executed in the order they are called, not in the order they are defined. Functions are defined once, but can be called (with different parameter values) as many times as needed.

After the s `main(),` function is defined, the program then defines `printInfo()` and then defines `getInfo()`. Up to that point, there is no input, no output, and no function has been called or executed – yet. The last line of code calls `main()`. A function is called using its name and any parameter values required by its definition. The main function has no parameters, therefore the parentheses after the word *main* are empty.

When `main()` is called, the code inside of main (distinguished by indentation) executes. The code inside of `main()` is within the **scope** of `main()` - it belongs to, is accessed by, and can be altered only by the `main()` function.
The code inside of the `main()` function first gathers input about the number of Employees, whose information will be entered. It then loops that number of times. During each entrance into the `while` loop, a call to the function `getInfo()` is made. Notice that `getInfo()` also does

not have any parameters. The `getInfo()` function gathers input from the user. It collects Employee name, ID, phone number, and age. It stores each of those pieces of information into four variables, namely *n* (for name), *id* (for the ID), *ph* (for the phone number), and *a* (for the age). The last statement in `getInfo()` is a function call to `printInfo(n,id,ph,a)`. Notice that the function, `printInfo(n,id,ph,a)`, has four parameters defined. As such, when `printInfo(n,id,ph,a)` is called, four values must be included inside the parentheses of the function call. These values will become the values of the function parameters for that function call. Every time a function with parameters is called, the parameter value can be different.

At first, this can seem a bit confusing. The most common question at this point is why the parameters for the function definition are different than those of the function call. To answer this question, let us first redefine some terms.

A **function** is a grouped-together collection of reusable code that can take inputs, offer outputs, return values, and perform tasks. When a task, method, or calculation must be performed more than once, a function can be created. That function can then be **called** anytime (and as many times) as the task must be performed. So, for example, in the program above, it is possible to input several Employee's information. For each Employee, information must be gathered, and information must be printed.

Therefore, for each Employee, the functions `getInfo( )` and `printInfo(n,id,ph,a)` will be called on. For `printInfo(n,id,ph,a)` to print out the information for each employee, the parameters `n,id,ph,` and `a` will be individual for each employee.

Function definitions are like templates, the **parameters** in the function call are very much like "place holders" (local variables). The function `printInfo(n,id,ph,a)` has four parameters. The first is called `n`, which will hold the first value sent to the function. So if the function is called with following parameters, `printInfo("John Smith",id,ph,a)`, then inside of the function `printInfo(n,id,ph,a)`, the value of `n` is set to `John Smith`.

Of course, most names are different and so the parameter, `n`, can and will hold any name sent to it. Notice that inside the function called, `getInfo()`, the Employee name is collected as input and stored in the local function variable called, `n`. In addition, inside of `getInfo()`, the Employee ID is collected and stored in a variable called, `id`, the Employee phone number is collected and stored in the variable called, `ph`, and finally, the age of the Employee is stored in a variable called, `a`.

Therefore, when `getInfo()` makes the function call `printInfo(n, id, ph, a)` it is sending it the information for name, ID, phone, and age that it just collected. When `printInfo(n, id, ph, a)` is called, the function matches up the parameter values from the call to the parameter variables in the definition. As such, the value in *n* will be stored in the variable *name*, the value in *id* will be stored in the variable *ID*, the value in *ph* will be stored in the variable *phone*, and the value in *a* will be stored in the variable *age*.

Example 5.2.2: Variables and Scope

Each function has its own **local variables**. These variables are used by and belong to only the function in which they are defined (their scope). When a function with parameters is called, the order and location of the parameters sent will determine the values of the local function parameter variables.

In addition, the values sent to the function parameter variables are stored in the parameter variables. The names of the parameter variables have no relationship to the names of variables in other functions (even if they are the same name). The following program will illustrate both of these important but unintuitive concepts.

```
def main():

    name1 ="John"
    name2= "Bob"
    name3= "Annie"
    SayHello(name1,name1, name1)


def SayHello(n1, name2, name3):

    print("Hello ", n1)
    print("Hello ", name2)
    print("Hello ", name3)

main()
```

The output for the above program is:

```
Hello  John
Hello  John
Hello  John
```

Why is this the output?

The main function is called first. Inside of the main function, three variables (local to main) are defined. Inside of main, `name1="John", name2 ="Bob",` and `name3 ="Annie"`. Next, from inside of the main function, the function SayHello is called. Note that the function SayHello is defined with three parameters. Pay special attention to the call to the SayHello function, `SayHello(name1,name1, name1)`, specifically, it is called with name1, name1, name1. In other words, the name John is sent to the SayHello function as all three parameters.

This means that the SayHello function parameter variables, `n1, name2,` and `name3`, all take on the value of John. This is a critical concept. The variable, `name3`, in the SayHello function definition is not the same variable, `name3`, that is in main. They are different variables with different locations in memory. This might be the same as saying that just because someone else in the world has your same name, they are not you.

For this reason, the output of the SayHello function is:

```
Hello  John
Hello  John
Hello  John
```

To better understand this important concept, make a few changes to this program and see what those changes do.

Example 5.2.3: Calling functions with parameter values

This next example will illustrate several ways to call a function with parameters.

```
def main():
    num1, num2, num3, num4 = eval(input("Enter four numbers
separated by commas: "))

    AddNumbers(num1,num2,num3,num4)
    AddNumbers(3, 6, 9, 12)
    AddNumbers(num1+1, num2+2, num3*4, num4**2)

#Definition of function called AddNumbers.
def AddNumbers(a,b,c,d):
    s=a+b+c+d
    print("The sum is ",s)

main()
```

Possible input/output:

```
Enter four numbers separated by commas: 1, 2, 3, 4
The sum is  10
The sum is  30
The sum is  34
```

In this example, the function called, `AddNumbers(a,b,c,d)`, has four parameter variables. When this function is called in the main function, notice that it is first called as

`AddNumbers(num1,num2,num3,num4)`, where *num1*, *num2*, *num3*, and *num4* hold the user inputs for these values. All four values will be sent to the function and therefore, within the function, *a = num1, b = num2, c = num3, d=num4*. The output is, *The sum is 10*.

Next, `AddNumbers(3, 6, 9, 12)` is called. In this case, within the function, *a = 3, b = 6, c = 9, d = 12*. The output is, *The sum is 30*. Finally, AddNumbers(num1+1, num2+2, num3*4, num4**2) is called from main. In this case, *a = num1+1, b=num2+2, c=num3*4, and d=num4**2*. The output is, *The sum is 34*.

This example illustrates the idea that a function can be called as many times as needed. It must be called with the number of parameters it is defined with (unless a parameter is given a default value). When calling a function, the call can be made with values, variables that contain values, or expressions (such as num4**2).

Example 5.2.4: An example of function utilization

The following program pulls together many of the concepts covered to this point, including functions, functions with parameters, nested decision structures, and user input and output. Review, type in, and run the following function. Then, make changes to evaluate the results of those changes.

```
# MortgageFunctions.py
# by Ami Gates
# This program will illustrate the nesting of for/in and if/else

def main():
    print("This program will display the monthly mortgage payments for four
different rates, ")
    print("and for three duration options.\n")
    homecost=eval(input("Please enter the house price: "))
    downpay=eval(input("Please enter the down payment: "))
    maxpay=eval(input("Please enter the maximum amount you wish to pay each
month: "))
    Mortgage(homecost,downpay,maxpay)


##----Definition of the Mortgage function -------------------
def Mortgage(hc, dp, maxpay):

    mortcost = hc - dp

    for yearrate in [.03, .035, .045, .05]:
        for years in [15, 20, 30]:
            mrate = yearrate/12
            months = years*12

            monthlypayment = mortcost*((mrate*((1+mrate)**months))/(((1 +
mrate)**months) - 1))
            yearpercent=yearrate*100
```

```
            if round(monthlypayment,2) <= maxpay:
                print("The monthly payment for an interest rate of ",
round(yearpercent,2), "%")
                print("and for ", years, "years")
                print("will be :", round(monthlypayment,2), "\n")
            else:
                print("The rate of ", round(yearpercent,2), "% combined with
",years, "years, does not match")
                print("your budget constraints.\n\n")



main()
```

A possible input/output for the above function:

```
     Please enter the house price: 250000

     Please enter the down payment: 50000

     Please enter the maximum amount you wish to pay each month:
1500
     The monthly payment for an interest rate of  3.0 %
     and for  15 years
     will be : 1381.16

     The monthly payment for an interest rate of  3.0 %
     and for  20 years
     will be : 1109.2

     The monthly payment for an interest rate of  3.0 %
     and for  30 years
     will be : 843.21

     The monthly payment for an interest rate of  3.5 %
     and for  15 years
     will be : 1429.77

     The monthly payment for an interest rate of  3.5 %
     and for  20 years
     will be : 1159.92

     The monthly payment for an interest rate of  3.5 %
     and for  30 years
     will be : 898.09

     The rate of  4.5 % combined with  15 years, does not match
     your budget constraints.
```

```
The monthly payment for an interest rate of  4.5 %
and for  20 years
will be : 1265.3

The monthly payment for an interest rate of  4.5 %
and for  30 years
will be : 1013.37

The rate of  5.0 % combined with  15 years, does not match
your budget constraints.


The monthly payment for an interest rate of  5.0 %
and for  20 years
will be : 1319.91

The monthly payment for an interest rate of  5.0 %
and for  30 years
will be : 1073.64
```

Program Example Review by Line of Code:

```
1. # MortgageFunctions.py
2. # by Ami Gates
3. # This program will illustrate the nesting of for/in and if/else

4. def main():
5.    print("This program will display the monthly mortgage payments
for four different rates, ")
6.    print("and for three duration options.\n")
7.    homecost=eval(input("Please enter the house price: "))
8.    downpay=eval(input("Please enter the down payment: "))
9.    maxpay=eval(input("Please enter the maximum amount you wish to
pay each month: "))
10.   Mortgage(homecost,downpay,maxpay)

11.   def Mortgage(hc, dp, maxpay):

12.         mortcost = hc - dp

13.         for yearrate in [.03, .035, .045, .05]:
14.               for years in [15, 20, 30]:
15.                     mrate = yearrate/12
```

```
16.                   months = years*12

17.                   monthlypayment =
mortcost*((mrate*((1+mrate)**months))/(((1 + mrate)**months) - 1))

18.                   yearpercent=yearrate*100

19.                   if round(monthlypayment,2) <= maxpay:
20.                         print("The monthly payment for an interest
rate of ", round(yearpercent,2), "%")
21.                         print("and for ", years, "years")
22.                         print("will be :", round(monthlypayment,2),
"\n")
23.                   else:
24.                         print("The rate of ", round(yearpercent,2),
"% combined with ",years, "years, does not match")
25.                         print("your budget constraints.\n\n")



26. main()
```

Code lines 1 – 3:

The first three lines of code are comments. They offer the program name, the author, and the goal of the program.

Code line 4:

Line 4 is the definition for the function called main. This function has no parameters.

Code lines 5 and 6:

Lines 5 and 6 print out information about what the program does. This is information for the user and is output.

Code lines 7 - 9:
These three lines of code collect information from the user, including homecost, downpay, and maxpay.

Code line 10:
This line of code makes a call to the function called Mortgage, with the required three parameters. The order of the parameters does matter. When calling the function, the homecost (called hc in the parameter list) must be first, followed by the downpayment (called dp), and then the maximum payment desired (called maxpay).

Code line 11:
This line of code is the function definition for the function called Mortgage. Here, the function is defined requiring three parameters. With the scope of the Mortgage function, the parameter variable names are hc (which will hold the value of house cost), dp (which will hold the value of down payment), and maxpay (which will hold the value of the maximum amount the buyer is willing to pay per month).

As a side note, this information should also be part of the program as comments.

Code line 12:
This line of code determines the actual total mortgage cost; the amount that will be borrowed. This is calculated by subtracting the down payment from the house cost.

Code lines 13 and 14:
Line 13 is the outer for loop and line 14 is the inner for loop. The outer for loop will loop through four interest rates, .03, .035, .045, and .05. The inner for loop (for each one of the interest rates) will loop through three duration options, 15, 20, and 30 years.

In other words, these two nested for loops will run for the following combinations:

Rate: .03, Duration: 15
Rate: .03, Duration: 20
Rate: .03, Duration: 30

Rate: .035, Duration: 15
Rate: .035, Duration: 20
Rate: .035, Duration: 30

Rate: .045, Duration: 15
Rate: .045, Duration: 20
Rate: .045, Duration: 30

Rate: .05, Duration 15
Rate: .05, Duration 20
Rate: .05, Duration 30

For each outer loop, the inner loop repeats three times (once for each rate).

Code lines 14 - 17:
These lines of code are inside the scope of the inner for loop. Together, they calculate the monthly rate (mrate), the number of months (months), and the monthly payment. The monthly payment uses the formula needed to calculate mortgage payments.

Code line 18:

This line of code converts the yearly interest rate to a percentage (such as converting .03 to 3). This will be used in the print statements that follow.

Code lines 19 - 25:
Line 19 is an if type decision structure nested inside of the inner for loop. Recall that the user inputs the maximum amount he/she wishes to pay each month. This if statement checks to see if the current monthly payment is less than or equal to that user's defined upper bound. If it is, the information, including the monthly payment, the interest rate, and the number of years (duration of mortgage) is output for the user (lines 20-22). If the monthly payment is too large (exceeds the user maximum), the program outputs a message that this combination of rate and duration exceeds the budget constraints (lines 23 – 35).

Code line 26:
Calls the main function and starts the program.

It is recommended as an exercise to rewrite the above program to use different interest rates or durations. It is also recommended that the reader create a flowchart of this program to better understand the flow of the nested for loops and the contained if/else decision structure.

5.3: Function Return Values

Functions have three primary parts. First, the **function definition** in which the local variables and local activities for the function are defined. Second, the optional **parameters**, which are zero or more pieces of information, in the form of variables, strings, numbers, or even other functions, that are passed directly to the function via the **parameter list**. Third, the optional **return** value(s) of the function. A function can return a value, or other structure, such as a list (which is covered in the next chapter). A function can have zero parameters and zero return values. This section will investigate functions that return values and structures. Some of the structures noted in this section will be covered in more detail in future chapters.

Example 5.3.1: Function example with a return value

```
def main():
    firstnum = eval(input("Please enter a number: "))
    secondnum = eval(input("Please enter a second number: "))
    thirdnum = eval(input("Please enter a third number: "))

    x = Calc(firstnum,secondnum,thirdnum)
    print("The product is: ",x)

def Calc(fnum,snum,tnum):

    sum_of_three_nums = fnum+snum+tnum
    prod_of_three_nums= fnum*snum*tnum
```

```
        print("The sum of the three numbers is: ", sum_of_three_nums)
        return  prod_of_three_nums

    main()
```

An output from the above program is:

```
Please enter a number: 2

Please enter a second number: 4

Please enter a third number: 6
The sum of the three numbers is:  12
The product is:  48
```

The above program illustrates a return statement in a function. Inside of the main function is the statement,

```
x = Calc(firstnum,secondnum,thirdnum)
```

This statement calls the function called `Calc` with three parameters. The variable `x` is set equal to the function call. The variable `x` will capture and store what the function returns. The return statement in the function is the following:

```
return  prod_of_three_nums
```

Once `prod_of_three_nums` is calculated in the `Calc` function, it is sent back to the main function using the return statement. Therefore, `x` will equal the value in `prod_of_three_nums`. For this reason, when `x` is printed in the main function, it prints the value of the product.

Functions can return values, variables, or structures, such as lists or tuples. While data types and structures will be discussed in greater detail in future chapters, the following example offers a look at two return options.

Example 5.3.2: Structure return options for functions

The following program contains two functions. The first requires parameters and returns a tuple (two values). The second function does not require parameters and returns a list. Review and run the following program. Make small changes to see how these changes affect the output. Below the program code is a possible input and output for this program

```
 def main():
```

```
    [f,l]=GetName()
    print("\nYour name is: ", f, " ", l)
    firstnum = eval(input("Please enter a number: "))
    secondnum = eval(input("Please enter a second number: "))
    thirdnum = eval(input("Please enter a third number: "))

    x,y = Calc(firstnum,secondnum,thirdnum)
    print("The sum is: ",x)
    print("The  product is: ",y)

def Calc(fnum,snum,tnum):

    sum = fnum+snum+tnum
    prod= fnum*snum*tnum

    return  sum,prod

def GetName():

    firstname=input("Enter your first name: ")
    lastname=input("Enter your last name: ")

    return [firstname,lastname]

main()
```

Possible input and output for the above program:

```
Enter your first name: John

Enter your last name: Smith

Your name is:  John   Smith

Please enter a number: 2

Please enter a second number: 4

Please enter a third number: 6
The sum is:  12
The  product is:  48
```

5.4: Optional and Default Function Parameters

What if parameters are missing when a function is called? What if a programmer wishes to have the option of not requiring the parameters, or of giving the parameters default values? There are several methods for creating and calling functions.

First, recall that when a function is defined with parameters, the order of the parameters matters. For example, the following function definition statement

```
def MyFunc(a, b, c):
```

would require three parameters when called. One method for calling this function might be using the following statement;

```
MyFunc(1,2,3)
```

This call assumes that `a` will be 1, `b` will be 2, and `c` will be 3 because of the order.

However, it is possible to call a function using the **names** of the parameters. For example, the following is also permissible:

```
MyFunc(c=3,a=1,b=2)
```

Because the names of the parameters are used in the call to the function, the order does not matter. While it is not recommended, there are also blends of these methods. For example, the following statement is also a feasible function call:

```
MyFunc(1,c=3,b=2)
```

This will work because the value of `a` is assumed to be 1 and is in the correct order. The values of `b` and `c` are named and so their order does not matter. That being said, this next statement will **not** work and will cause a syntax error:

```
MyFunc(c=3,1,b=2) ### Not correct – syntax error
```

Here, the intension is that `a` is 1, `b` is 2, and `c` is 3. However, because the `a` is not named and the "1" is not in the correct location, an error will occur.

Overall, clarity is best. Either use the function parameter order, or use all parameter names.

Functions can also be defined with **default parameter values** which allows them to be absent from the function call. However, the order of the parameters still matters. In most cases, if parameters are to be left out, the names of the parameters are used in the call to avoid logical or syntax errors. As with the above case, there are many variations of these themes. Often, the best option is clarity. The following function definition uses a default value. It is placed at the end so that it can easily be left out of the call if desired.

```
def MyFunc2(a, b, c = 3):
```

When this function is called, there are many options. The following three calls are all feasible:

```
MyFunc2(1,2)
MyFunc2(b=2, a=1)
MyFunc2(1,2,7)
```

Test some of these cases by creating a small program. What happens in each case?

Example 5.4.1: Default function parameters, order, and calling parameters by name

The following program collects three numbers from the user. It also defines a function called `AddThree`. The `AddThree` function has four parameters, all with default values. Within the `main` function, the `AddThree` function is called seven times with different parameters. Review the program and the output. Make changes and additions to see what they do.

```
def main():

    firstnum = eval(input("Please enter a number: "))
    secondnum = eval(input("Please enter a second number: "))
    thirdnum = eval(input("Please enter a third number: "))

    AddThree()
    AddThree(firstnum)
    AddThree(firstnum, secondnum)
    AddThree(firstnum, secondnum, thirdnum)
    AddThree(firstnum, secondnum, thirdnum, "John")
    AddThree(f = firstnum, t=thirdnum)
    AddThree(name="john", t=thirdnum)


def AddThree(f=0, s=0, t=0, name="None"):
    sum = f + s + t
    if name != "None":
        print("Hello ", name)

    print("Sum of numbers sent is ", sum)


main()
```

Here is an input/output for this program:

```
Please enter a number: 2

Please enter a second number: 4
```

```
Please enter a third number: 6

Sum of numbers sent is  0
Sum of numbers sent is  2
Sum of numbers sent is  6
Sum of numbers sent is  12
Hello  John
Sum of numbers sent is  12
Sum of numbers sent is  8
Hello  john
Sum of numbers sent is  6
```

Example 5.4.2: Calling functions with optional parameters and returning values

As seen above, functions can be called with optional parameters assuming they are defined with parameter defaults. Functions can also be called with parameters that are not in order if the parameters are called by name.

In addition, functions can return information using the return statement. Functions can return lists, tuples, values, variables, and other data structures. The following program offers a further example of these options.

```
def main():
    firstnum = eval(input("Please enter a number: "))
    secondnum = eval(input("Please enter a second number: "))
    thirdnum = eval(input("Please enter a third number: "))

    Call1 = Calc2(3)
    Call2  = Calc2(s=secondnum, f=firstnum, num=5, name="Bob", t=11 )

    print("The first returned value in Call1 is: ", Call1[0])
    print("The first returned value in Call1 is: ", Call1[1])
    print("The first returned value in Call2 is: ", Call2[0])
    print("The first returned value in Call2 is: ", Call2[1])

def Calc2(f, s=10, t=25, num=5, name="Anne"):
    sum4 = f + s + t + num
    prod4= f * s * t * num
    return (sum4,prod4)

main()
```

In the program example above, the `Calc2` function is called twice. In the `main` function, information is gathered from the user and stored in the three variables, `firstnum`, `secondnum`, and `thirdnum`. Next, there are two function calls made to `Calc2`.

The `Calc2` function definition takes five parameters. Four of these parameters have default values and can be passed using their names, or not at all, if proper order is maintained. The first parameter, however, does not have a default and so must be passed, and passed first, in all calls.

The `Calc2` function calculates the sum of the four numbers and the product of the four numbers. The `return` statement returns a **tuple** of both results, as `(sum4, prod4)`. It could also have returned the values as a list as well. Lists and tuples will be covered in detail in Chapter 7, along with other data structures.

Inside of `main` there are two function calls to `Calc2`. An interesting feature of these two function calls is that they are each assigned to a variable:

```
Call1 = Calc2(3)
Call2  = Calc2(s=secondnum, f=firstnum, num=5, name="Bob",t=11 )
```

The variable `Call1` will be assigned the return value of the function call `Calc2(3)`. Similarly, `Call2` will be assigned the return value of the second call to `Calc2`. To step through this program and to better understand the return values, review the following input/output and the steps below.

Example input and output for the program:

```
Please enter a number: 10

Please enter a second number: 20

Please enter a third number: 30
The first returned value in Call1 is:  43
The first returned value in Call1 is:  3750
The first returned value in Call2 is:  46
The first returned value in Call2 is:  11000
```

Program Example Review by Steps:

Step 1: Enter `main`. As a note, the `main` function is defined first, and then the `Calc2` function is defined. Recall that a function is not executed until it is called. In this program, the last line of code calls `main`.

Step 2: Once inside of `main`, the first three statements collect information (numbers) from the user and store each in *`firstnum`*, *`secondnum`*, and *`thirdnum`*, respectively.

Step 3: A function call to `Calc2` (from inside of the main function) is made. This call is made with one parameter (the number 3). When the `Calc2` function is called with only "3" as a

parameter, the result of this call is that all other parameters of `Calc2` will default (so s = 10, t = 25, num = 5, and name= "Anne").

Step 4: Inside of `Calc2,` the local variable *sum4* is assigned the sum of the local variables: *f, s, t, num.* Therefore, in the output above, *sum4 = 3 + 10 + 25 + 5 = 43.* Similarly, the local variable *prod4* is assigned the product of the four parameter values.

Step 5: A tuple containing *sum4* and *prod4* is returned to the caller. Note that the call originated from inside of *main* via the statement: `Call1 = Calc2(3)`. Therefore, *Call1* is equal to the tuple *(43, 3750)*. Once the return statement returns the tuple to the caller, the next line of code after the call is then executed. In other words, once a function ends, the overall program execution returns to the next line of code after the function call, and continues.

Step 6: The next line of code to be executed is: `Call2 = Calc2(s=secondnum, f=firstnum, num=5, name="Bob", t=11 )`. Here again, the function `Calc2` is being called, and the return value will be placed in the variable called `Call2`. This function call is sending parameters to the `Calc2` function using the **naming method.** When variables are specifically named using the names in the function definition, the order of the parameters no longer matters. This call will result in the following assignments: `s = secondnum, f = firstnum, num =5, name = "Bob",` and `t = 11`.

Step 7 : Inside of `Calc2` and given the above value assignments and user input from the example input and output, `sum4` = *10 + 20 + 11 + 5* = `46`, and `prod4` = *10 * 20 * 11 * 5* = `11000`. Both of these values can be confirmed in the output above. As a reminder, the function `Calc2` can be called as many times as needed. Each call can generate a different result depending on the parameters it is called with.

Step 8: A tuple containing *sum4* and *prod4* is returned so that the caller assigns this tuple to the variable in `main` called `Call2`. In other words, after this return, *Call2 = (46, 11000)*.

Step 9: Once the function returns, the execution of the code resumes at the next statement, which is a print statement in `main`. The last four statements in `main` are all print statements that output the results of the calculations made and returned by `Calc2`.

Step 10: Execute all four print statements and output results to the user.

To print the first value in the tuple, the square bracket method is used. `Call1[0]` is the first value in the tuple that is assigned to `Call1` and `Call1[1]` is the second value in the tuple. This same method can be used for lists and strings and will be discussed in detail in a following chapter.

```
print("The first returned value in Call1 is: ", Call1[0])
```

The above print statement will print the first value in the `Call1` tuple (which is 43). Specifically, `Call1 = (43, 3750)` and so `Call1[0]` is the value 43 and `Call1[1]` is the value 3750.

In summary, the above example illustrates methods for calling functions with parameters, as well as returning (and accessing) results using a tuple. As an interesting consideration, a function can contain multiple return statements (as desired), but once a return statement is executed, the function ends. As such, multiple return statements might be placed inside of if/else decision structures, allowing for the determination at **run-time** of which return statement to execute.

Example 5.4.3: Combining Functions, Loops, and Decision Structures

This next example will combine many of the concepts covered to this point. To practice, type in the program, save it, and run it. Try a few inputs and review the outputs. Next, evaluate each statement by hand – trace the program step by step – to see what it does. Notice the while loop, the decision structures, function calls, and return values. Try to make minor changes to the program to see what happens.

```
# Exercise.py
# by Ami Gates
# This program assists with exercise management

def main():
    keepgoing="yes"
    print("Welcome to the Exercise and Calories Program")
    name=input("Please enter your name: ")
    print("Hi ", name, "to calculate calories related to exercise, some
information is needed. ")

    while keepgoing == "yes":
        weight=eval(input("Please enter your current weight in lbs: "))
        gender=input("Please enter your gender as M or F: ")

        print("Exercise burns calories! Please choose the exercise that you
want to learn about.")
        exercise = input("You can type walk or run: ")
        time=eval(input("Please enter the amount of time in minutes you will
perform this exercise: "))

        if gender=="M" or gender=="Male" or gender=="male" or gender=="m":
            result=getCalsMale(exercise,time, weight)
            print("Given your information, you will burn *approximately* ",
round(result,2), "calories by", exercise," for ", time, "minutes")

        else:
            result=getCalsFemale(exercise,time, weight)
            print("Given your information, you will burn *approximately* ",
round(result,2), "calories by", exercise," for ", time, "minutes")
```

```
        keepgoing=input("To quit, type Quit, otherwise press enter to
continue: ")
        #print(keepgoing)

def getCalsMale(exer, mins, lbs):
    if exer=="walk" or exer=="walking":
        print("Walking briskly is great exercise!")
        if lbs<=150:
            totalcals=(200*mins)/60
        elif lbs > 150:
            totalcals=(200*mins)/60

    if exer=="run" or exer=="running":
        print("Running is great exercise!")
        if lbs<=150:
            totalcals=(530*mins)/60
        elif lbs > 150:
            totalcals=(650*mins)/60
    return totalcals

def getCalsFemale(exer, mins, lbs):
    if exer=="walk" or exer=="walking":
        print("Walking briskly is great exercise!")
        if lbs<=140:
            totalcals=(200*mins)/60
        elif lbs > 140:
            totalcals=(240*mins)/60

    if exer=="run" or exer=="running":
        print("Running is great exercise!")
        if lbs <= 140:
            totalcals=(570*mins)/60
        elif lbs > 140:
            totalcals=(750*mins)/60
    return totalcals

main()
```

Input/output example for the above program:

```
Welcome to the Exercise and Calories Program

Please enter your name: Benny
Hi  Benny to calculate calories related to exercise, some information is
needed.

Please enter your current weight in lbs: 145

Please enter your gender as M or F: male
Exercise burns calories! Please choose the exercise that you want to learn
about.

You can type walk or run: run
```

```
Please enter the amount of time in minutes you will perform this exercise: 45
Running is great exercise!
Given your information, you will burn *approximately*  397.5 calories by run
for  45 minutes

To quit, type Quit, otherwise press enter to continue: Quit
```

The above example program collects information from the user, such as weight, gender, and exercise preference. It then uses these values to determine which function to call. The function called will return the total calories burned, given the user's information and duration of the activity. The program illustrates a while loop that runs until the user types "Quit" (and anything other than "yes", in fact). The program also illustrates the use of the if/elif decision structures within the while loop and within the function definitions. The two functions each return the total calories burned so that the program can print out the results.

At this point, you should be comfortable with these concepts and with writing programs that utilize these tools. Keep in mind that writing programs is a creative activity. In the same way that knowing how to write in English does not make you a poet, knowing basic syntax does not yet make you a coder. However, practice and patience will!

5.5: An Introduction to Scope in Python

Up to this point, we have been writing relatively small programs and functions. As such, we have not had to worry too much about what scope is, why it matters, and how it works in Python (as all languages have uniqueness). Scope generally refers to where and when a given variable is defined, and when and where it can be updated. Python uses **indentation** to note scope (or ownership). Here is a small example to illustrate these ideas:

```
# Scope1.py
# by Ami Gates

def main():
    a = 3
    b= 4
    c = 5
    print("Inside of main, a=",a,"b=",b,"c=",c)

    somefunction(a,b,c)

    print("Inside of main after call 1 to somefunction, a=",a,"b=",b,"c=",c)

    a=somefunction(a,b,c)

    print("Inside of main after call 2 to somefunction, a=",a,"b=",b,"c=",c)

def somefunction(r,s,t):
    a=r+10
    b=s+10
```

```
    c=t+10

    print("Inside of somefunction, a=",a,"b=",b,"c=",c)
    return a

main()
```

The output from this program is:

```
Inside of main, a= 3 b= 4 c= 5
Inside of somefunction, a= 13 b= 14 c= 15
Inside of main after call 1 to somefunction, a= 3 b= 4 c= 5
Inside of somefunction, a= 13 b= 14 c= 15
Inside of main after call 2 to somefunction, a= 13 b= 4 c= 5
```

In the above example, there are two functions, `main` and `somefunction`. Each of these two functions has its own "space" in memory. Neither function will automatically share parameters, variables, or information, even if parameter or variable names happen to be the same.

Inside of `main`, three variables are defined and assigned values, *a = 3*, *b = 4*, and *c= 5*. To view these variables and their values, a print statement, `print("Inside of main, a=",a,"b=",b,"c=",c)` , follows.
Next, the function `somefunction` is called from within `main`. Notice that the definition of `somefunction` requires that it is called with three parameters (also called arguments). Inside of the definition of `somefunction` the variable names, *a*, *b*, and *c* are used again, such that: *a = r+10*, *b =s+10*, and *c= t+10*. Inside of the function `somefunction`, the values of *a*, *b*, and *c* are updated. Does this update affect the values of *a*, *b*, and *c* inside of *`main`* (or any other part of the program)? The answer is no.

Within `main` the print statement,

```
     print("Inside of main after call 1 to somefunction,
a=",a,"b=",b,"c=",c),
```

illustrates that the values of *a*, *b*, and *c* inside of `main` remain the same even after `somefunction` is called. They have not been affected by anything that has taken place in `somefunction`. The result of this print statement is: `Inside of main after call 1 to somefunction, a= 3 b= 4 c= 5.`

To be clear, variables inside of the scope of a function are specific to that function. If two or more functions define a variable with the same name, they are in fact different variables with different memory locations. They are not related.

For functions to share information, they may receive parameter values and they may return values. For example, in the above example program, the statement in `main`,

```
a=somefunction(a,b,c)
```

allows the variable *a* defined in `main` to collect the result generated by *`somefunction`*. For this to work, *`somefunction`* must have a **return** statement that returns the value that it wants the `main` function to have for *`a`*.

Again, the key here is that each function, including `main`, has its own space in memory and its own variables (even if variable names are reused). Changing the value of a variable inside of one function, will have no effect on the value of any variables inside of any other function. ?To transfer information between functions, and in so doing, to update variable values, parameters and return statements must be employed.? The body of a function and all contained variables is the **scope** of that function. Python uses indentation to signify scope. The following two examples illustrate scope and naming.

Example 5.5.1: Functions, scope, and variable naming

```
# Scope2.py
# by Ami Gates

def main():
    a=1
    b=2
    c=3
    print("The values of a, b, and c in main are: ", a, b, c)
    f1()
    print("The values of a, b, and c in main are: ", a, b, c)
    f2()
    print("The values of a, b, and c in main are: ", a, b, c)


def f1():
    a = 34
    b = 25
    c = "bob"
    print("The values of a, b, and c in f1() are: ", a, b, c)

def f2(a=12.6,b=13.7, c=11):
    print("The values of a, b, and c in f2() are: ", a, b, c)

main()
```

The output for the above program is:

```
The values of a, b, and c in main are:  1 2 3
The values of a, b, and c in f1() are:  34 25 bob
The values of a, b, and c in main are:  1 2 3
```

```
The values of a, b, and c in f2() are:  12.6 13.7 11
The values of a, b, and c in main are:  1 2 3
```

The above example illustrates that the variables, *a*, *b*, and *c* are unique in each function and are not affected between functions. The next example will build on this example using the addition of return statements so that values can be shared between functions.

Example 5.5.2: Sharing variable values with the return statement

```
# Scope3.py
# by Ami Gates

def main():
    a=1
    b=2
    c=3
    print("The values of a, b, and c in main are: ", a, b, c)
    (a,b,c) = f1()
    print("The values of a, b, and c in main are: ", a, b, c)
    (a,b,c) = f2()
    print("The values of a, b, and c in main are: ", a, b, c)


def f1():
    a = 34
    b = 25
    c = "bob"
    print("The values of a, b, and c in f1() are: ", a, b, c)
    return (a,b,c)

def f2(a=12.6,b=13.7, c=11):
    print("The values of a, b, and c in f2() are: ", a, b, c)
    return (a,b,c)

main()
```

The output for the above program is:

```
The values of a, b, and c in main are:  1 2 3
The values of a, b, and c in f1() are:  34 25 bob
The values of a, b, and c in main are:  34 25 bob
The values of a, b, and c in f2() are:  12.6 13.7 11
The values of a, b, and c in main are:  12.6 13.7 11
```

Here, the values of *a*, *b*, and *c* are returned and so assigned to the variables *a*, *b*, and *c* inside of `main`. This occurs after each function call.

Scope can create unexpected program outcomes, especially for large programs. Scope can be more complicated when functions are nested (contained within) other functions.

5. 6: Nested Functions

This section is a little bit more advanced but is very important. It will cover function nesting, as well as an introduction to how variables are stored and accessed in memory.

Throughout this book, we have, and will continue to use **variables** to store information. Recall that a variable can be thought of as a name or label for a location in memory. Computer memory does not actually use "human English words" to name its locations. However, having a name or word that makes sense to the programmer is part (a nice part) of high level coding.

The location in memory that is named by a variable name can hold a value. A simple variable definition and assignment might be the following statement:

```
PersonAge = 28
```

This statement creates a memory location, names it PersonAge (from our point of view), and places the value 28 into its memory location. Note that this is a simplification of the process, but will suffice for now. The value 28, in this case, can be accessed by using the variable name, `PersonAge`. So, if we were to print `PersonAge`, the number 28 would be printed.

The scope examples in the above section illustrated that variables defined within a function may not be known to, accessible to, or updatable by a different function. Of course, there are some interesting exceptions to this, which include functions defined inside of other functions (called **nested functions**). The words, **global** and **local** are sometimes used to discuss the range or accessibility of a given variable.

Overall, the variables in any program have certain locations where they are known and accessible. This location is called their scope. If a variable is defined in the `main` function, it is called, **global to main**, as it is accessible anywhere within `main` and by any function defined inside of `main`.

However, if a variable is defined inside of a function that is defined inside of `main`, that variable is **local** to that function and it not known to main. Consider the follow program:

```
# NestedScope.py
# by Ami Gates

#-Inside main() ----------------------------
def main():
    a=1
    b=2
```

```
    c=3

    #---define f1 inside of main (nested)----------------
    def f1():
        b = 25
        c = "bob"
        print("The values of a, b, and c in f1() are: ", a, b, c)
        return c
    # ---end of f1 definition----------------------------

    print("The values of a, b, and c in main are: ", a, b, c)

    c = f1()
    print("The values of a, b, and c in main are: ", a, b, c)

    c = f2()
    print("The values of a, b, and c in main are: ", a, b, c)

#--END OF main() definition-----------------------------------

#--Define f2 ------------
def f2():
    a=45
    b=55
    c=65
    print("The values of a, b, and c in f2() are: ", a, b, c)
    return c
# end of f2 defintion

# Call main here to start program
main()
```

The output for the above program is:

```
The values of a, b, and c in main are:  1 2 3
The values of a, b, and c in f1() are:  1 25 bob
The values of a, b, and c in main are:  1 2 bob
The values of a, b, and c in f2() are:  45 55 65
The values of a, b, and c in main are:  1 2 65
```

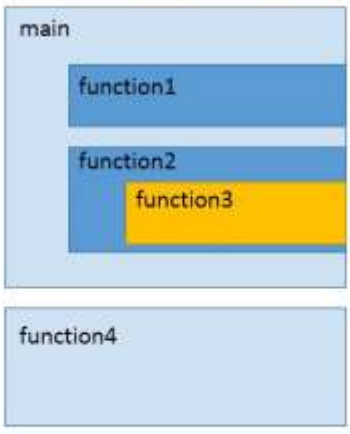
In the above program, the function called `f1` is defined inside of `main`. Inside of `f1`, the variables, *b*, and *c* are redefined. However, the variable *a* is not defined in `f1`. Because `f1` is inside of `main`, it shares the values of the variables defined in `main`. This creates an interesting situation.

What are the values of *a*, *b*, and *c*, inside of function `f1`? Notice that when the values of *a*, *b*, and *c* are printed from inside of function `f1`, that variable *a* retains the value that it was assigned in `main`. The variable *a* is global to all of `main`. However, the variables *b* and *c*, while also global

and defined in `main` are redefined in function `f1`. Therefore, locally and inside of `f1`, the variable *a* still has the value 1, but the variable *b* now has the value 25, and the variable *c* now has the value "bob".

The function called `f2` is defined outside of `main`. Because of this, `f2` does not share any variable definitions that are global in the `main` function. Inside (local) to function `f2`, the values are *a, b,* and, *c* are: 45, 55, and 65. The function `f2` returns its local value for *c* to the `main` function. This alters the value of *c* in `main`. However, *a* and *b*, remain the same in `main`.

The key idea here is that any function defined inside of another function (**nested function**) automatically shares its parent-function's scope. Therefore, the values of the parent-function variables are known to the nested function. A nested function can locally define or redefine any variable values and can use new or updated variables inside of its own function (scope) that will have no effect on its parent variables. Specifically, a definition or redefinition of a variable inside of a function will not affect the global value of that variable outside of the nested function (unless it is returned and redefined in the parent function). Figure 5.1 offers a visual example of scope.



Scope:
1) function1 is inside of main and within the scope of main.
2) function2 is inside of main and within the scope of main, but is not within the scope of function1
3) function3 is inside of main and within the scope of main. It is also within the scope of function2. It is not in the scope of function1
4) function 4 has its own scope and is not within the scope of main or the functions. None of the other functions are within the scope of function4.

Figure 5.1: Visual illustration of functional scope.

A First Look at Importing Modules and `__name__`

Python allows all users to create separate Python files or modules (.py) and to import these modules into other programs. In other words, and as a very simple example, it is possible to write

a program that adds two numbers together. Call this program Adds.py. Next, write a small program that subtracts two numbers. Call this program Subs.py. Then, import the Subs.py module into the Adds.py program and look at the value of the special variable called __name__.

```
#Subs.py
#Author Ami Gates

print("In the Subs.py program the value of __name__ is ", __name__)
def main():
    x=3
    y=4
    diff=x-y
    print(diff)

main()
```

```
#Adds.py
#Author Ami Gates
import Subs

print("In the Adds.py program the value of __name__ is ", __name__)
def main():
    x=1
    y=2
    sum=x+y
    print(sum)

main()
```

When the program Adds.py is run, the output is:

```
In the Subs.py program the value of __name__ is  Subs
-1
In the Adds.py program the value of __name__ is  __main__
3
```

When the Subs module is imported into Adds.py, the code in Subs.py is run first. When the code in Subs.py is run, the __name__ of the module is Subs and the value of the difference of x and y is printed. Then, after Subs is run, the next line of code in Adds.py commences. In this case, inside of Adds.py, the value of __name__ is __main__ because the main function is running in Adds.py.

More precisely in the above example, the "__main__" is the scope name for the Adds.py program's top level of execution. In general, a module (program's) "__name__" is assigned the value "__main__" when the program is run as a script or from standard input. When the module Subs is imported and run, the value of "__name__" within module Subs.py is "__Subs__" (not __main__).

One common use for checking the value of "__name__" is the prevention of auto-running an imported module. In the above example, the module (program) called Subs.py (that was imported into Adds.py) ran as soon as Adds.py was run. However, it might be that case that we do not want Subs to run right away and that instead we want to run it later in the Adds program. The following programs illustrate this idea.

```
#Adds.py
#Author Ami Gates

#Now, Subs will not run because when Subs is imported, __name__
# is not equal to __main__ it is equal to __Subs__
import Subs

def main():
    print("In the Adds.py program the value of __name__ is ", __name__)
    x=1
    y=2
    sum=x+y
    print("The sum is ",sum)
    #Here we can call the main function in Subs and print
    #the returned values
    value=Subs.main()
    print("The difference from Subs.py is ", value)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

```
#Subs.py
#Author Ami Gates

def main():
    x=3
    y=4
    diff=x-y
    return diff

# This stops Subs from being auto-run when imported into Adds.py
if __name__ == "__main__":
    # execute only if run as a script - not when imported as a module
    main()
```

The output for the Adds.py program is:

```
In the Adds.py program the value of __name__ is  __main__
The sum is  3
The difference from Subs.py is  -1
```

Here, when Adds.py imports Subs, the Subs program does not automatically run. Rather, within the Adds program, the Subs.main() function is called when needed. The use of the statements,

```
if __name__ == "__main__":
# execute only if run as a script – not when imported as a module
    main()
```

allows the program to call functions in an imported module when they are needed, rather than having the module run upon import.

In the next chapter, we will look at further details, such as data types, strings, and data structures. While we have already seen some of these concepts, it is now time to better understand each one and to review the rules and methods.