FUNCTIONS

5.1 Introduction to Functions

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

Most programs perform tasks that are large enough to be broken down into several subtasks. For this reason, programmers usually break down their programs into small manageable pieces known as functions.
Instead of writing a large program as one long sequence of statements, it can be written as several small functions, each one performing a specific part of the task. These small functions can then be executed in the desired order to perform the overall task.

This approach is sometimes called ***divide and conquer*** because a large task is divided into several smaller tasks that are easily performed.

When using functions in a program, you isolate each task within the program in its own function.

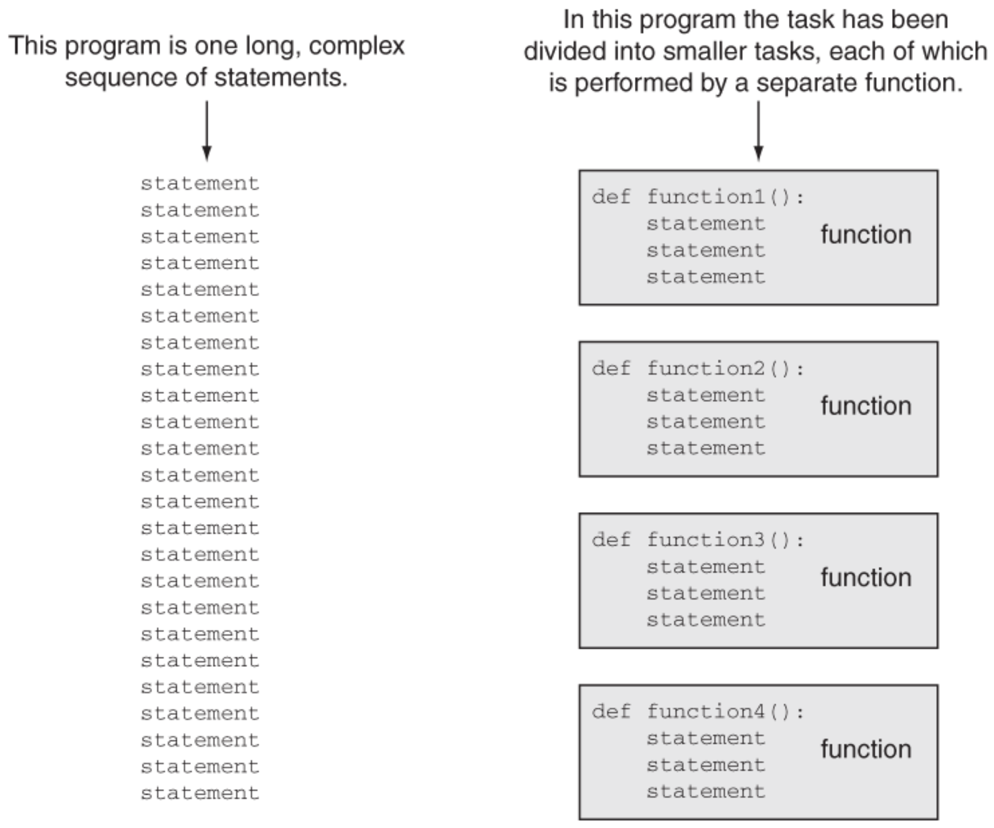
A program that has been written with each task in its own function is called a *modularized program.*

Void Functions and Value-Returning Functions

A *void function*, executes the statements it contains and then terminates.

A *value-returning function*, executes the statements that it contains, and then it returns a value back to the statement that called it.

Using Functions



This program is one long, complex sequence of statements.
In this program the task has been divided into smaller tasks, each of which is performed by a separate function.
statement
def function1():
def function2():
def function3():
def function4():
function

Figure 5-1 Using functions to divide and conquer a large task

Benefits of Modularizing a Program with Functions

- Simpler Code
- Code Reuse
- Better Testing
- Faster Development
- Easier Facilitation of Teamwork

5.2 Defining and Calling a Void Function

The code for a function is known as a function definition. To execute the function, you write a statement that calls it.

Defining a Function

To create a function you write its *definition*.

The general format of a function definition in Python:

```
def function_name():
    statement
    statement
    etc.
```

Example:

```
def message():
    print('I am Arthur,')
    print('King of the Britons.')
```

This code defines a function named message. The message function contains a block with two statements.

Calling a Function

A function definition specifies what a function does, but it does not cause the function to execute. To execute a function, you must *call* it.

The general format of a function call in Python:

    function_name()

Example:

    message()

When a function is called, the interpreter jumps to that function and executes the statements 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. When this happens, we say that the function *returns*.

We can define many functions in a program. It is common for a program to have a **main** function that is called when the program starts. The main function then calls other functions in the program as they are needed. It is often said that the main function contains a program's *mainline logic,* which is the overall logic of the program.

Indentation in Python

In Python, each line in a block must be indented. The last indented line after a function header is the last line in the function's block. When you indent the lines in a block, make sure each line begins with the same number of spaces. Otherwise an error will occur.

Note: Names of functions follow the same rules as names of variables:

- 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 or A through Z, the digits 0 through 9, or underscores.
- Uppercase and lowercase characters are distinct.

5.3 Designing a Program to Use Functions

Programmers commonly use a technique known as **top-down design** to break down an algorithm into functions.

Flowcharting a Program with Functions

A function call is shown with a rectangle that has vertical bars at each side. The name of the function that is being called is written on the symbol.

Example:

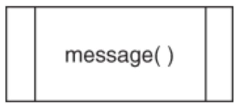
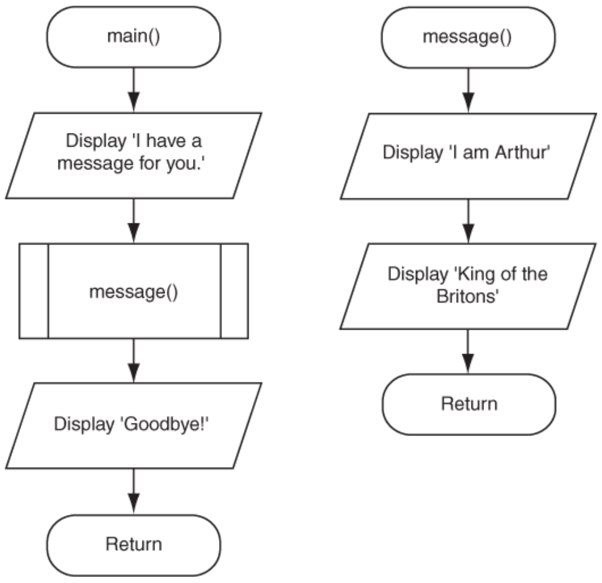


Figure 5-8 Function call symbol

Flowchart: for each function in a program a separate flowchart is usually drawn.



Top-down design of programs that uses functions

A technique known as *top-down design* breaks down an algorithm into functions. The process of top-down design is performed in the following manner:

- The overall task that the program is to perform is broken down into a series of subtasks.
- Each of the subtasks is examined to determine whether it can be further broken down into more subtasks. This step is repeated until no more subtasks can be identified.
- Once all of the subtasks have been identified, they are written in code.

The top-down design begins by looking at the topmost level of tasks that must be performed and then breaks down those tasks into lower levels of subtasks.

Hierarchy Charts

Hierarchy charts, also known as *structure charts,* are used to give a visual representation of the relationships between functions.

A hierarchy chart shows boxes that represent each function in a program. The boxes are connected in a way that illustrates the functions called by each function.

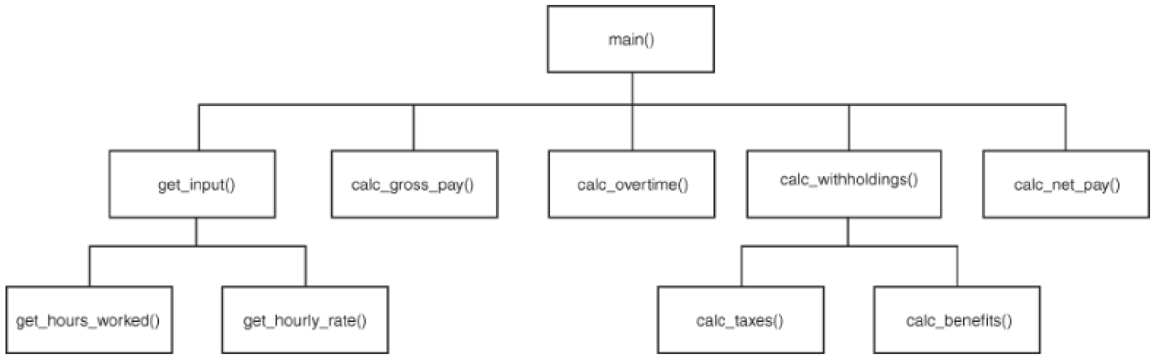


Figure 5-10 A hierarchy chart

The hierarchy chart does not show the steps that are taken inside a function, they do not replace flowcharts or pseudocode.

5.4 Local 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.

Scope and Local Variables

A variable's *scope* is the part of a program in which the variable may be accessed. A variable is visible only to statements in the variable's scope. A local variable's scope is the function in which the variable is created.

In addition, a local variable cannot be accessed by code that appears inside the function at a point before the variable has been created.

Because a function's local variables are hidden from other functions, the other functions may have their own local variables with the same name.

5.5 Passing Arguments to Functions

An argument is any piece of data that is passed into a function when the function is called. A parameter is a variable that receives an argument that is passed into a function.

Sometimes it is useful 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. Here is an example of a function that has a parameter variable:

```
def show_double(number):
    result = number * 2
    print(result)
```

This function's name is show_double. Its purpose is to accept a number as an argument and display the value of that number doubled. The word number appears inside the parentheses. This is the name of a parameter variable. This variable will be assigned the value of an argument when the function is called.

Parameter Variable Scope

A parameter variable's scope is the function in which the parameter is used. All of the statements inside the function can access the parameter variable, but no statement outside the function can access it.

Passing Multiple Arguments

Often it's useful to write functions that can accept multiple arguments.

Making Changes to Parameters

When an argument is passed to a function in Python, the function parameter variable will reference the argument's value.

Any changes that are made to the parameter variable will not affect the argument.

The form of argument passing that is used in Python, where a function cannot change the value of an argument that was passed to it, is commonly called *pass by value*.

Keyword Arguments

Arguments are passed by position to parameter variables in a function. In addition to this conventional form of argument passing, the Python language allows you to write an argument in the following format, to specify which parameter variable the argument should be passed to:

parameter_name=value

In this format, parameter_name is the name of a parameter variable and value is the value being passed to that parameter. An argument that is written in accordance with this syntax is known as a *keyword argument*.

Mixing Keyword Arguments with Positional Arguments

It is possible to mix positional arguments and keyword arguments in a function call, but the positional arguments must appear first, followed by the keyword arguments.

Example:

```
    show_interest(10000.0, rate=0.01, periods=10)

def show_interest(principal, rate, periods)
```

In this statement, the first argument, 10000.0, is passed by its position to the principal parameter. The second and third arguments are passed as keyword arguments.

5.6 Global Variables and Global Constants

A global variable is accessible to all the functions in a program file.

When a variable is created by an assignment statement inside a function, the variable is local to that function. Consequently, it can be accessed only by statements inside the function that created it. When a variable is created by an assignment statement that is written outside all the functions in a program file, the variable is *global*. A global variable can be accessed by any statement in the program file, including the statements in any function.

If you want a statement in a function to assign a value to a global variable, you must declare the global variable in the function.

5.7 Introduction to Value-Returning Functions: Generating Random Numbers

A value-returning function is a function that returns a value back to the part of the program that called it. Python, as well as most other programming languages, provides a library of prewritten functions that perform commonly needed tasks. These libraries typically contain a function that generates random numbers.

A *value-returning function* is a special type of function. It is like a void function in the following ways.

- It is a group of statements that perform a specific task.
- When you want to execute the function, you call it.

When a value-returning function finishes, however, it returns a value back to the part of the program that called it. The value that is returned from a function can be used like any other value: it can be assigned to a variable, displayed on the screen, used in a mathematical expression (if it is a number), and so on.

Standard Library Functions and the import Statement

Python comes with a *standard library* of functions that have already been written.

These functions, known as *library functions,* perform many of the tasks that programmers commonly need to perform. We have already used several of Python's library functions: print, input, and range…

Some of Python's library functions are built into the Python interpreter.

To use built-in functions in a program, we call the function (the print, input, range, and other functions).

Many of the functions in the standard library are stored in files that are known as *modules*. These modules, which are copied to a computer when installing Python, help organize the standard library functions.

In order to call a function that is stored in a module, an import statement has to be written at the top of your program. An import statement tells the interpreter the name of the module that contains the function.

```
import (name of the module)
```

One of the Python standard modules is named math. The math module contains various mathematical functions that work with floating-point numbers. If you want to use any of the math module's functions in a program, you should write the following import statement at the top of the program:

```
import math
```

This statement causes the interpreter to load the contents of the math module into memory and makes all the functions in the math module available to the program.

Because you do not see the internal workings of library functions, many programmers think of them as *black boxes.*

Generating Random Numbers

Python provides several library functions for working with random numbers. These functions are stored in a module named random in the standard library. To use any of these functions you first need to write this import statement at the top of your program:

```
import random
```

This statement causes the interpreter to load the contents of the random module into memory. This makes all of the functions in the random module available to your program.

The random-number generating function: **randint**.

We need to use *dot notation* to refer to it.

Example:

```
number = random.randint (1, 100)
```

The part of the statement that reads random.randint(1, 100) is a call to the randint function. Arguments (1 and 100) tell the function to give an integer random number in the range of 1 through 100. (The values 1 and 100 are included in the range.)

The randint function returns an integer value.

Use of randint in a math expression.

Example:

```
x = random.randint (1, 10) * 2
```

A random number in the range of 1 through 10 is generated and then multiplied by 2. The result is assigned to the x variable.

The randrange, random, and uniform Functions

To use any of these functions you need to write import random at the top of your program.

The randrange function takes the same arguments as the range function. The randrange function returns a randomly selected value from a sequence of values.

Example:

The following statement assigns a random number in the range of 0 through 9 to the number variable:

```
number = random.randrange(10)
```

The argument, in this case 10, specifies the ending limit of the sequence of values. The function will return a randomly selected number from the sequence of values 0 up to, but not including, the ending limit. The following statement specifies both a starting value and an ending limit for the sequence:

```
number = random.randrange(5,10)
```

When this statement executes, a random number in the range of 5 through 9 will be assigned to number. The following statement specifies a starting value, an ending limit, and a step value:

```
number = random.randrange(0, 101, 10)
```

In this statement the randrange function returns a randomly selected value from the following sequence of numbers:

```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

Both the randint and the randrange functions return an integer number.

The random function

The random function returns a random floating-point number. You do not pass any arguments to the random function. When you call it, it returns a random floating point number in the range of 0.0 up to 1.0 (but not including 1.0).

Example:

```
number = random.random()
```

The uniform function

The uniform function returns a random floating-point number, but allows you to specify the range of values to select from. Here is an example:

Example:

```
number = random.uniform(1.0, 10.0)
```

In this statement the uniform function returns a random floating-point number in the range of 1.0 through 10.0 and assigns it to the number variable.

Random Number Seeds

The numbers that are generated by the functions in the random module are not truly random, they are actually *pseudorandom numbers* that are calculated by a formula. The formula that generates random numbers has to be initialized with a value known as a *seed value*. The seed value is used in the calculation that returns the next random number in the series. When the random module is imported, it retrieves the system time from the computer's internal clock and uses that as the seed value. The system time is an integer that represents the current date and time, down to a hundredth of a second.

If the same seed value were always used, the random number functions would always generate the same series of pseudorandom numbers.

There may be some applications in which we want to always generate the same sequence of random numbers. Then we can call the random.seed function to specify a seed value.

Example:

```
    random.seed(10)
```

The value 10 is specified as the seed value. If a program calls the random.seed function, passing the same value as an argument each time it runs, it will always produce the same sequence of pseudorandom numbers.

5.8 Writing Your Own Value-Returning Functions

A value-returning function has a return statement that returns a value back to the part of the program that called it.

Here is the general format of a value-returning function definition in Python:

```
def function_name():
    statement
    statement
    etc.
    return expression
```

The value of the *expression* that follows the key word return will be sent back to the part of the program that called the function. This can be any value, variable, or expression that has a value (such as a math expression).

Example of a value-returning function:

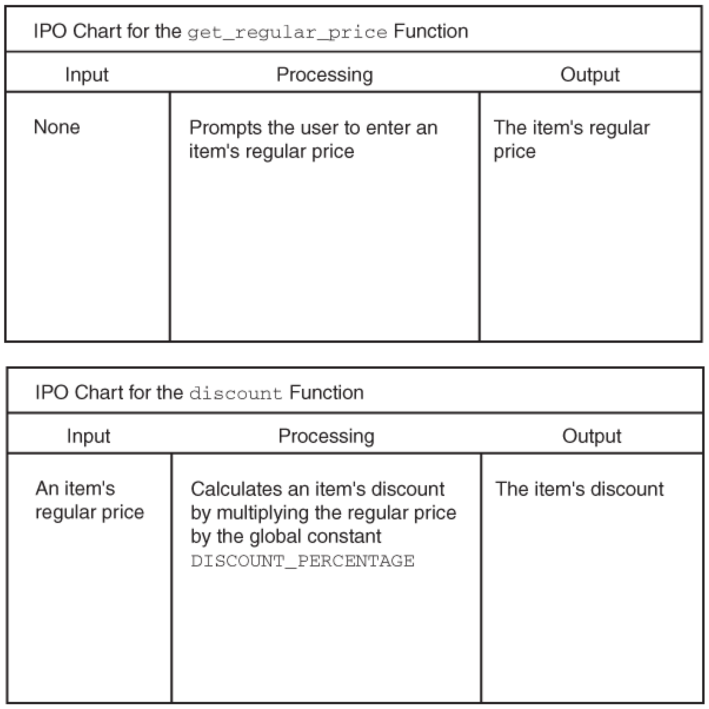
```
def sum(num1, num2):
    result = num 1 + num 2
    return result
```

Simplified:

```
def sum(num1, num2):
    return num 1 + num 2
```

Using IPO Charts

An IPO chart is a simple but effective tool that programmers sometimes use for designing and documenting functions. IPO stands for *input, processing,* and *output,* and an *IPO chart* describes the input, processing, and output of a function. These items are usually laid out in columns: the input column shows a description of the data that is passed to the function as arguments, the processing column shows a description of the process that the function performs, and the output column describes the data that is returned from the function.



IPO Chart for the `get_regular_price` Function		
Input	Processing	Output
None	Prompts the user to enter an item's regular price	The item's regular price

IPO Chart for the `discount` Function		
Input	Processing	Output
An item's regular price	Calculates an item's discount by multiplying the regular price by the global constant `DISCOUNT_PERCENTAGE`	The item's discount

Figure 5-25 IPO charts for the `getRegularPrice` and `discount` functions

Returning Strings

Functions can also return that return strings.

Example:

```
def get_name():
    # Get the user's name.
    name = input('Enter your name: ')
    # Return the name.
    return name
```

Returning Multiple Values

The examples of value-returning functions that we have looked at so far return a single value. In Python, however, you are not limited to returning only one value. You can specify multiple expressions separated by commas after the return statement, as shown in this general format:

return *expression1, expression2, etc.*

5.9 The math Module

The Python standard library's math module contains numerous functions that can be used in mathematical calculations.

These functions accept one or more values as arguments, perform a mathematical operation using the arguments, and return the result.

The functions return a float value, except the ceil and floor functions, which return int values.

Example:

```
result = math.sqrt(16)
```

The sqrt function accepts an argument and returns the square root of the argument. This statement calls the sqrt function, passing 16 as an argument.

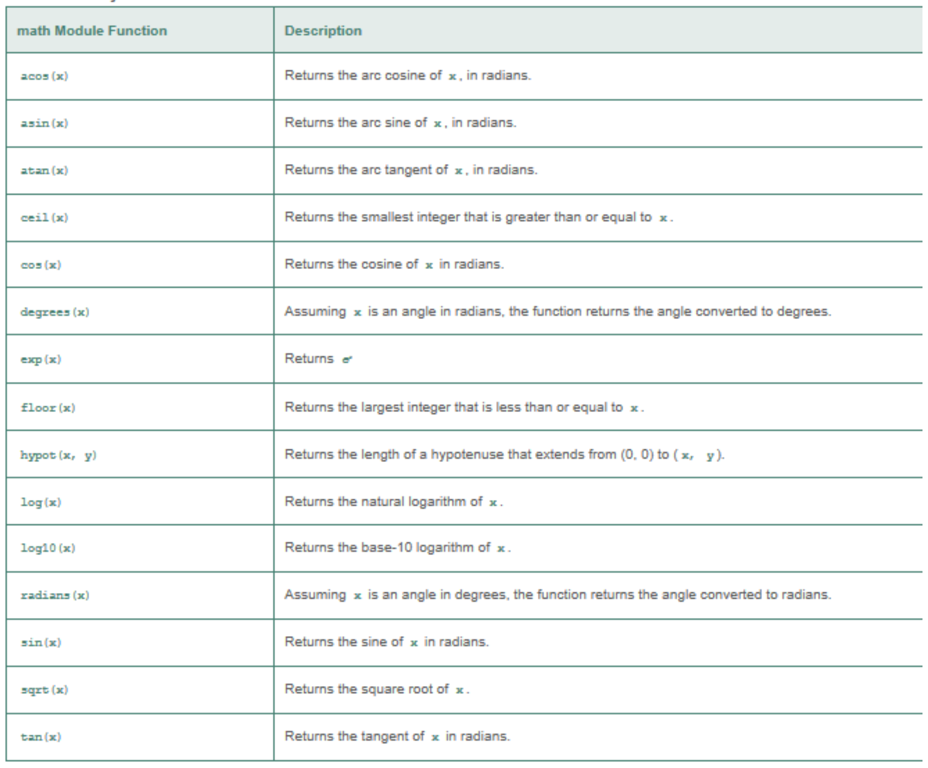


Table 5-2 Many of the functions in the `math` module

math Module Function	Description
`acos(x)`	Returns the arc cosine of `x`, in radians.
`asin(x)`	Returns the arc sine of `x`, in radians.
`atan(x)`	Returns the arc tangent of `x`, in radians.
`ceil(x)`	Returns the smallest integer that is greater than or equal to `x`.
`cos(x)`	Returns the cosine of `x` in radians.
`degrees(x)`	Assuming `x` is an angle in radians, the function returns the angle converted to degrees.
`exp(x)`	Returns e^x
`floor(x)`	Returns the largest integer that is less than or equal to `x`.
`hypot(x, y)`	Returns the length of a hypotenuse that extends from (0, 0) to (`x`, `y`).
`log(x)`	Returns the natural logarithm of `x`.
`log10(x)`	Returns the base-10 logarithm of `x`.
`radians(x)`	Assuming `x` is an angle in degrees, the function returns the angle converted to radians.
`sin(x)`	Returns the sine of `x` in radians.
`sqrt(x)`	Returns the square root of `x`.
`tan(x)`	Returns the tangent of `x` in radians.

The math.pi and math.e Values

The math module also defines two variables, pi and e, which are assigned mathematical values for *pi* and *e*. You can use these variables in equations that require their values. For example, the following statement, which calculates the area of a circle, uses pi. (Notice that we use dot notation to refer to the variable.)

```
area = math.pi * radius**2
```

5.10 Storing Functions in Modules

A module is a file that contains Python code. Large programs are easier to debug and maintain when they are divided into modules.

A large and complex program should be divided into functions that each performs a specific task. More functions in a program should be organized; the functions can be stored in modules.

A module is a file that contains Python code. When you break a program into modules, each module should contain functions that perform related tasks.

Example:

Let suppose that you are writing an accounting system. You would store:

- all of the account receivable functions in their own module;
- all of the account payable functions in their own module;
- all of the payroll functions in their own module.

This approach is called *modularization*, and makes the program easier to understand, test, and maintain.

Modules make easier to reuse the same code in more than one program. If you have written a set of functions that are needed in several different programs, you can place those functions in a module. Then, you can import the module in each program that needs to call one of the functions.

Menu-Driven Programs

A *menu-driven program* displays a list of the operations on the screen, and allows the user to select the operation that he or she wants the program to perform. The list of operations that is displayed on the screen is called a *menu.*

Once the user types a menu selection, the program uses a decision structure to determine which menu item the user selected.