Python Programming (16ITE01)

UNIT - II

T. PRATHIMA, DEPT. OF IT, CBIT

Syllabus

UNIT-II

Functions: Introduction, Defining and Calling a Function, Designing a Program to Use Functions, Local Variables, Passing Arguments to Functions, Global Variables and Global Constants, Value-Returning Functions Generating Random Numbers, Writing Our Own Value-Returning Functions, The math Module, Random Module, Time Module and Storing Functions in Modules.

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

Using functions to divide and conquer a large task

This program is one long, complex sequence of statements.

```
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
```

In this program the task has been divided into smaller tasks, each of which is performed by a separate function.

```
def function1():
    statement
    statement
    statement
```
function

```
def function2():
    statement
    statement
    statement
```
function

```
def function3():
    statement
    statement
    statement
```
function

```
def function4():
    statement
    statement
    statement
```
function

Benefits of Modularizing a Program with Functions

- **Simpler Code:**
  - 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.
- **Code Reuse:**
  - Functions also reduce 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 and then be executed any time it is needed.
  - This benefit of using functions is known as *code reuse* because you are writing the code to perform a task once and then reusing it each time you need to perform the task.
- **Better Testing:**
  - 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.
  - This makes it easier to isolate and fix errors.

Benefits of Modularizing a Program with Functions

- **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 a 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.

- **Easier Facilitation of Teamwork:**
  - Functions also make it easier for programmers to work in teams.
  - When a program is developed as a set of functions that each performs an individual task, then different programmers can be assigned the job of writing different functions.

Void Functions and Value-Returning Functions

- When you call a *void function*, it simply executes the statements it contains and then terminates.
- When you call a *value-returning function*, it executes the statements that it contains, and then it returns a value back to the statement that called it.
- The input function is an example of a value-returning function.
- When you call the input function, it gets the data that the user types on the keyboard and returns that data as a string.
- The int and float functions are also examples of value-returning functions.
- You pass an argument to the int function, and it returns that argument's value converted to an integer.
- Likewise, you pass an argument to the float function, and it returns that argument's value converted to a floating-point number.

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.

Function Names

- Just as you name the variables that you use in a program, you also name the functions.
- A function's name should be descriptive enough so that anyone reading your code can reasonably guess what the function does
- For example, a function that calculates gross pay might be named calculate_gross_pay.

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

Defining and Calling a Function

- To create a function you write its *definition*.
- Here is the general format of a function definition in Python:

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

Defining and Calling a Function

- The first line is known as the *function header*.
- It marks the beginning of the function definition.
- The function header begins with the key word def, followed by the name of the function, followed by a set of parentheses, followed by a colon.
- Beginning at 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.
- This indentation is required because the Python interpreter uses it to tell where the block begins and ends.

Example of a function


```
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.
- Executing the function will cause these statements to execute.

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.
- This is how we would call the message function:

```
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*.

Example

Program 5-1 (function_demo.py)

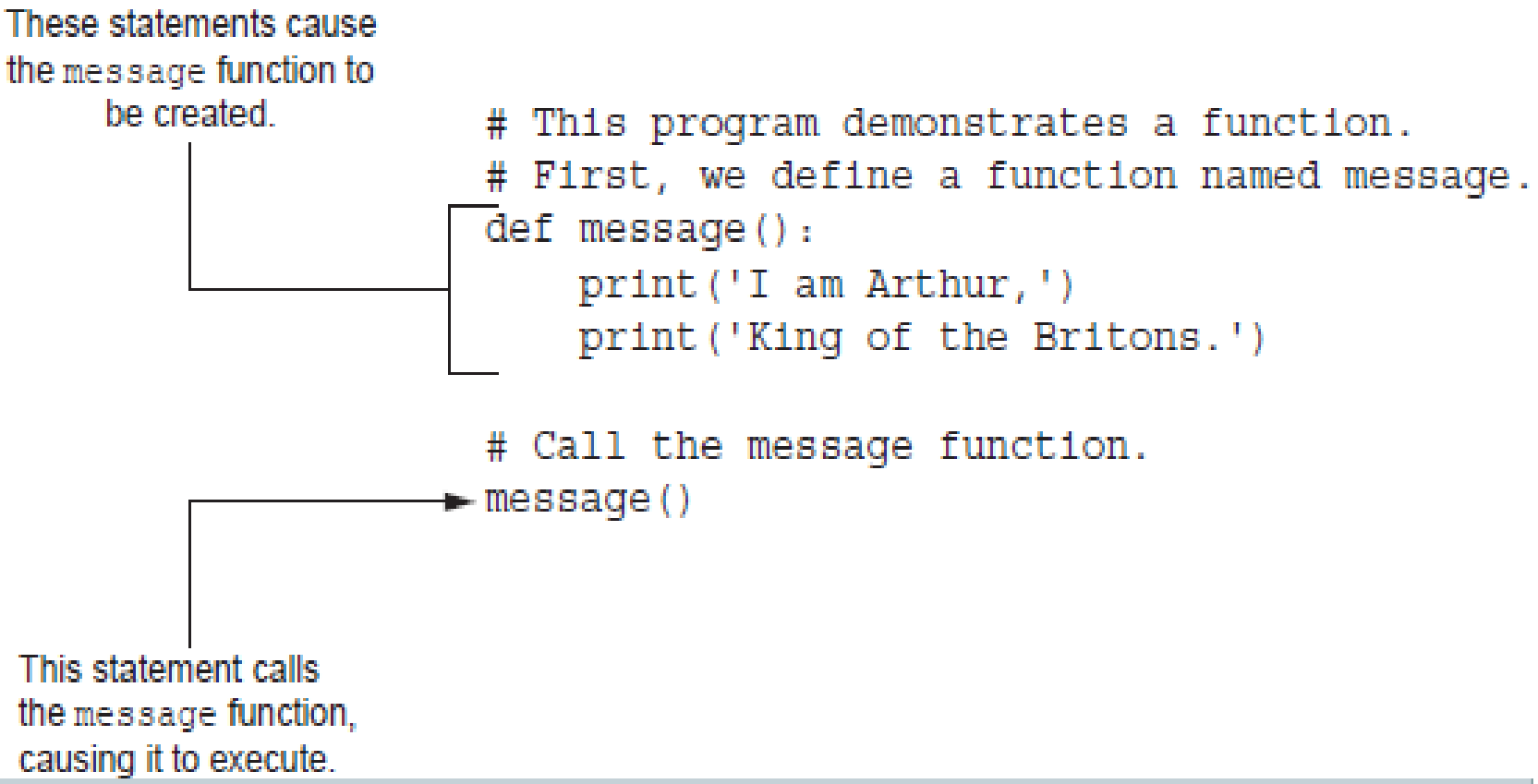
```
1  # This program demonstrates a function.
2  # First, we define a function named message.
3  def message():
4      print('I am Arthur,')
5      print('King of the Britons.')
6
7  # Call the message function.
8  message()
```

Program Output

```
I am Arthur,
King of the Britons.
```

The function definition and the function call

Figure 5-2 The function definition and the function call



main()

- It is possible to define many functions in a program.
- In fact, 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.

Example


```
1  # This program has two functions. First we
2  # define the main function.
3  def main():
4      print('I have a message for you.')
5      message()
6      print('Goodbye!')
7
8  # Next we define the message function.
9  def message():
10     print('I am Arthur,')
11     print('King of the Britons.')
12
13 # Call the main function.
14 main()
```

- **Program Output**

I have a message for you.
I am Arthur,
King of the Britons.
Goodbye!

Calling the main()

The interpreter jumps to the `main` function and begins executing the statements in its block.

```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

# Call the main function.
main()
```

The first statement in the `main` function calls the `print` function in line 4. It displays the string `'I have a message for you'`. Then, the statement in line 5 calls the `message` function. This causes the interpreter to jump to the `message` function, as shown in Figure 5-4. After the statements in the `message` function have executed, the interpreter returns to the `main` function and resumes with the statement that immediately follows the function call. As shown in Figure 5-5, this is the statement that displays the string `'Goodbye!'`.

Calling the message function

The interpreter jumps to the `message` function and begins executing the statements in its block.

```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

# Call the main function.
main()
```

The message function returns

5-5 The message function returns

When the message function ends, the interpreter jumps back to the part of the program that called it and resumes execution from that point.

```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

# Call the main function.
main()
```

That is the end of the main function, so the function returns as shown in Figure 5-6. There are no more statements to execute, so the program ends.

The main function returns

Figure 5-6 The `main` function returns

```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

# Call the main function.
main()
```

When the `main` function ends, the interpreter jumps back to the part of the program that called it. There are no more statements, so the program ends.

NOTE: When a program calls a function, programmers commonly say that the *control* of the program transfers to that function. This simply means that the function takes control of the program's execution.

Indentation in Python

The last indented line is the last line in the block.

```
def greeting():
    print('Good morning!')
    print('Today we will learn about functions.')
```

These statements are not in the block.

```
print('I will call the greeting function.')
greeting()
```

Indentation in Python

- When you indent the lines in a block, make sure each line begins with the same number of spaces.
- Otherwise an error will occur. For example, the following function definition will cause an error because the lines are all indented with different numbers of spaces.

```
def my_function():
        print('And now for')
    print('something completely')
        print('different.')
```

- In an editor there are two ways to indent a line:

  (1) by pressing the Tab key at the beginning of the line, or

  (2) by using the spacebar to insert spaces at the beginning of the line.

- You can use either tabs or spaces when indenting the lines in a block, but don't use both.
- Doing so may confuse the Python interpreter and cause an error.

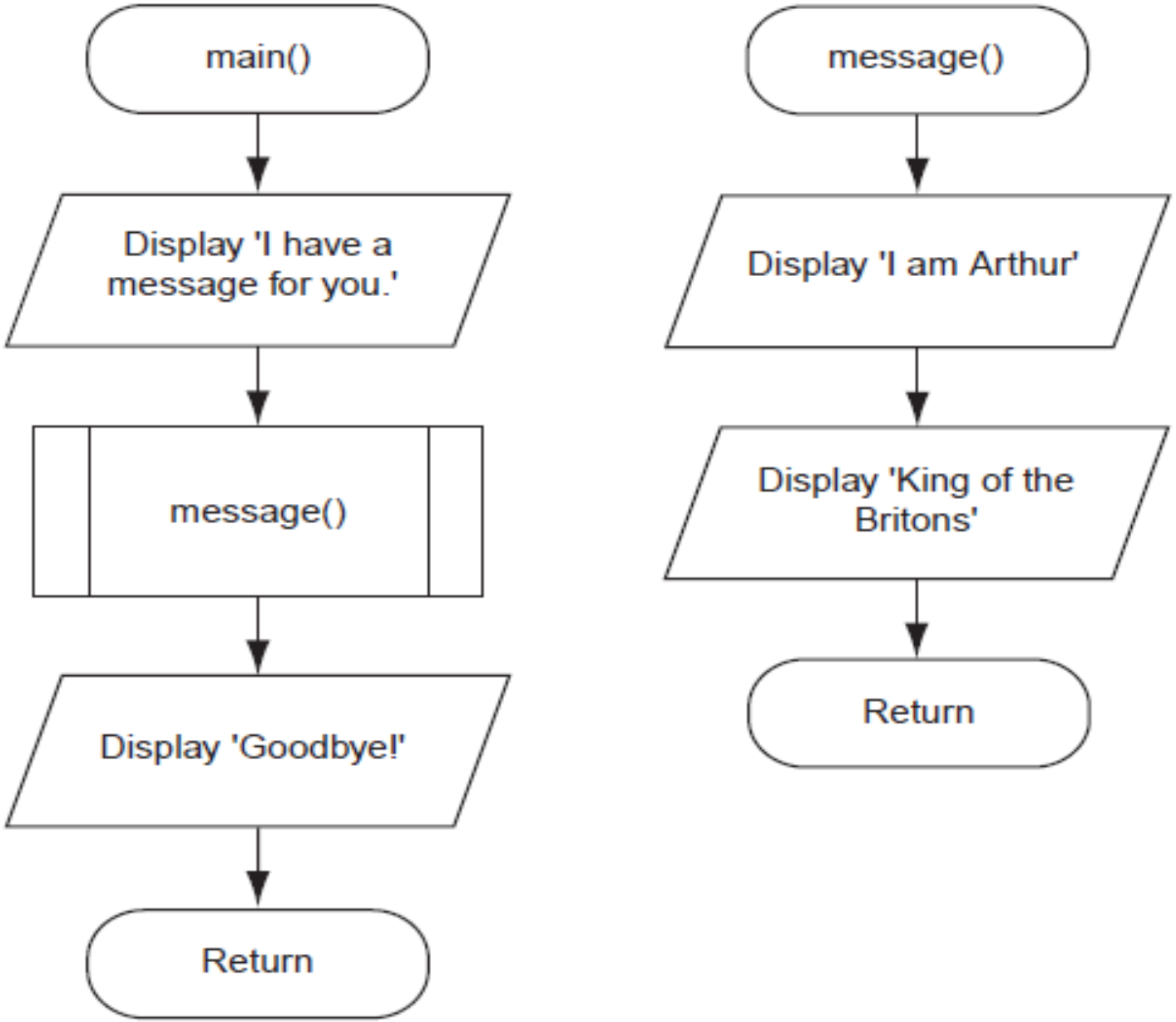
More about indenting....

- IDLE, as well as most other Python editors, automatically indents the lines in a block.
- When you type the colon at the end of a function header, all of the lines typed afterward will automatically be indented.
- After you have typed the last line of the block you press the Backspace key to get out of the automatic indentation.
- Python programmers customarily use four spaces to indent the lines in a block.
- You can use any number of spaces you wish, as long as all the lines in the block are indented by the same amount.
- Blank lines that appear in a block are ignored.

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



Top-Down Design

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

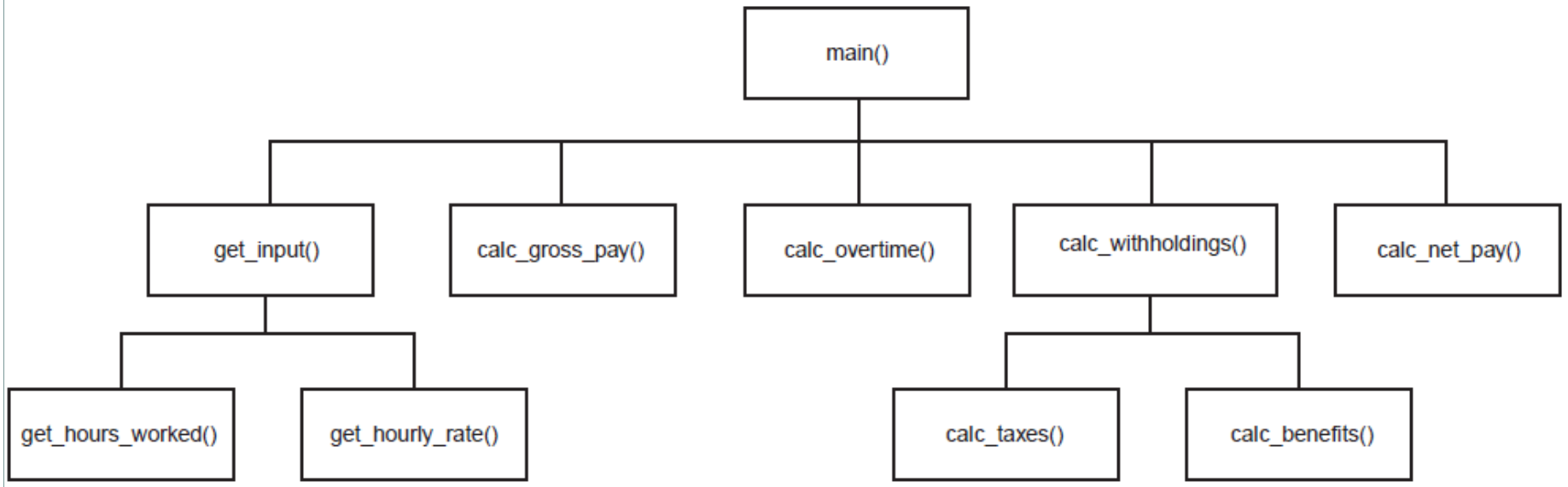
- The process of top-down design is performed in the following manner:
  - The overall task that the program is to perform is broken down into a series of subtasks.
  - Each of the subtasks is examined to determine whether it can be further broken down into more subtasks.
  - This step is repeated until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

- This process is called top-down design because the programmer 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

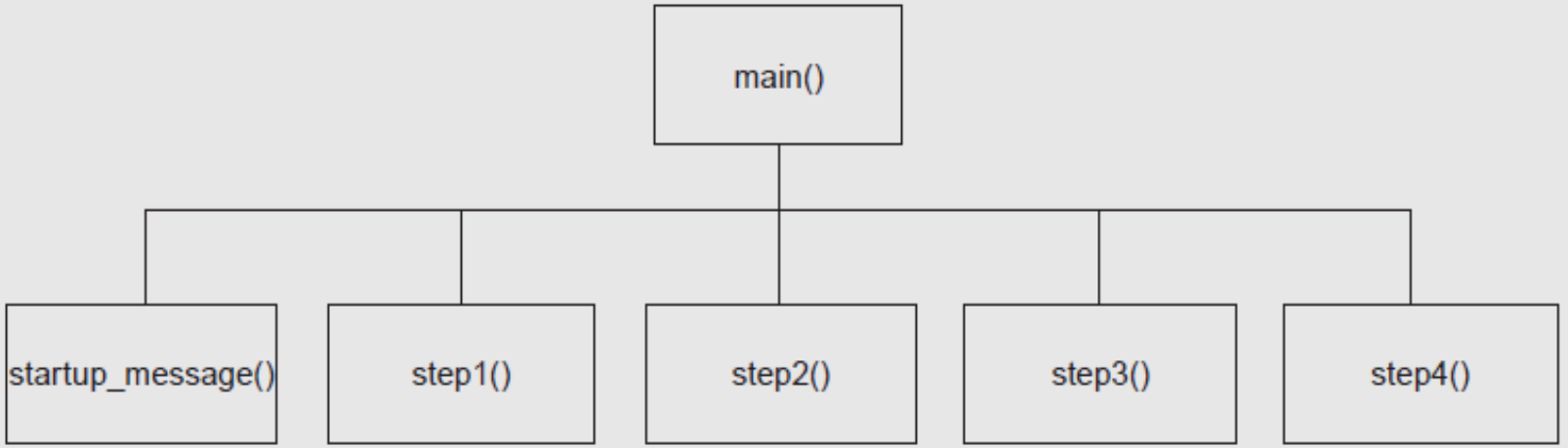
- Flowcharts are good tools for graphically depicting the flow of logic inside a function, but they do not give a visual representation of the relationships between functions.
- Programmers commonly use *hierarchy charts* for this purpose.
- A hierarchy chart, which is also known as a *structure chart*, shows boxes that represent each function in a program.
- The boxes are connected in a way that illustrates the functions called by each function.

A hierarchy chart



Defining and Calling Functions

Figure 5-11 Hierarchy chart for the program



As you can see from the hierarchy chart, the `main` function will call several other functions. Here are summaries of those functions:

- `startup_message`—This function will display the starting message that tells the technician what the program does.
- `step1`—This function will display the instructions for step 1.
- `step2`—This function will display the instructions for step 2.
- `step3`—This function will display the instructions for step 3.
- `step4`—This function will display the instructions for step 4.

```
# This program displays step-by-step instructions
# for disassembling an Acme dryer.
# The main function performs the program's main logic.
def main():
    # Display the start-up message.
    startup_message()
    input('Press Enter to see Step 1.')
    # Display step 1.
    step1()
    input('Press Enter to see Step 2.')
    # Display step 2.
    step2()
    input('Press Enter to see Step 3.')
    # Display step 3.
    step3()
    input('Press Enter to see Step 4.')
    # Display step 4.
    step4()

# The startup_message function displays the
# program's initial message on the screen.
def startup_message():
    print('This program tells you how to')
    print('disassemble an ACME laundry dryer.')
    print('There are 4 steps in the process.')
    print()

# The step1 function displays the instructions
# for step 1.
def step1():
    print('Step 1: Unplug the dryer and')
    print('move it away from the wall.')
    print()
```

```
# The step2 function displays the instructions
# for step 2.
def step2():
    print('Step 2: Remove the six screws')
    print('from the back of the dryer.')
    print()

# The step3 function displays the instructions
# for step 3.
def step3():
    print('Step 3: Remove the back panel')
```

Program 5-3 *(continued)*

```
46      print('from the dryer.')
47      print()
48
49  # The step4 function displays the instructions
50  # for step 4.
51  def step4():
52      print('Step 4: Pull the top of the')
53      print('dryer straight up.')
54
55  # Call the main function to begin the program.
56  main()
```

Program Output

```
This program tells you how to
disassemble an ACME laundry dryer.
There are 4 steps in the process.

Press Enter to see Step 1. Enter
Step 1: Unplug the dryer and
move it away from the wall.

Press Enter to see Step 2. Enter
Step 2: Remove the six screws
from the back of the dryer.

Press Enter to see Step 3. Enter
Step 3: Remove the back panel
from the dryer.

Press Enter to see Step 4. Enter
Step 4: Pull the top of the
dryer straight up.
```

Pausing Execution Until the User Presses Enter

- Sometimes you want a program to pause so the user can read information that has been displayed on the screen.
- When the user is ready for the program to continue execution, he or she presses the Enter key and the program resumes.
- In Python you can use the input function to cause a program to pause until the user presses the Enter key.

```
input('Press Enter to see Step 1.')
```

- This statement displays the prompt 'Press Enter to see Step 1.' and pauses until the user presses the Enter key.

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.
- Anytime you assign a value to a variable inside a function, you create a *local variable*.
- A local variable belongs to the function in which it is created, and only statements inside that function can access the variable.
- (The term *local* is meant to indicate that the variable can be used only locally, within the function in which it is created.)
- An error will occur if a statement in one function tries to access a local variable that belongsto another function.

Example

```
1  # Definition of the main function.
2  def main():
3      get_name()
4      print('Hello', name)       # This causes an error!
5
6  # Definition of the get_name function.
7  def get_name():
8      name = input('Enter your name: ')
9
10 # Call the main function.
11 main()
```

This program has two functions: `main` and `get_name`. In line 8 the `name` variable is assigned a value that is entered by the user. This statement is inside the `get_name` function, so the `name` variable is local to that function. This means that the `name` variable cannot be accessed by statements outside the `get_name` function.

The `main` function calls the `get_name` function in line 3. Then, the statement in line 4 tries to access the `name` variable. This results in an error because the `name` variable is local to the `get_name` function, and statements in the `main` function cannot access it.

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.

Example

- A local variable cannot be accessed by code that appears inside the function at a point before the variable has been created.
- For example, look at the following function.
- It will cause an error because the print function tries to access the val variable, but this statement appears before the val variable has been created.
- Moving the assignment statement to a line before the print statement will fix this error.

```
def bad_function():
    print('The value is', val) # This will cause an error!
    val = 99
```

Example

```
1  # This program demonstrates two functions that
2  # have local variables with the same name.
3
4  def main():
5      # Call the texas function.
6      texas()
7      # Call the california function.
8      california()
9
10 # Definition of the texas function. It creates
11 # a local variable named birds.
12 def texas():
13     birds = 5000
14     print('texas has', birds, 'birds.')
15
16 # Definition of the california function. It also
17 # creates a local variable named birds.
18 def california():
19     birds = 8000
20     print('california has', birds, 'birds.')
21
22 # Call the main function.
23 main()
```

Program Output

```
texas has 5000 birds.
california has 8000 birds.
```

Visibility of variables: Each function has its own birds variable

Each function has its own `birds` variable

```
def texas():
    birds = 5000
    print('texas has', birds, 'birds.')
```


```
def california():
    birds = 8000
    print('california has', birds, 'birds.')
```

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.
- 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

Program 5-6 (pass_arg.py)

```
1  # This program demonstrates an argument being
2  # passed to a function.
3
4  def main():
5      value = 5
6      show_double(value)
7
8  # The show_double function accepts an argument
9  # and displays double its value.
10 def show_double(number):
11     result = number * 2
12     print(result)
13
14 # Call the main function.
15 main()
```

Program Output

```
10
```

5-13 The value variable is passed as an argument

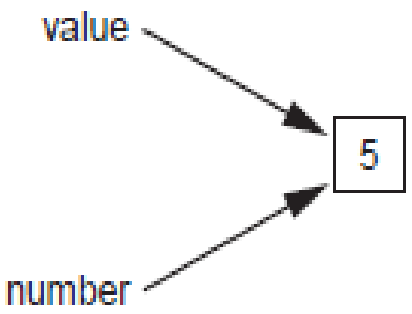
```
def main():
    value = 5
    show_double(value)


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

5-14 The value variable and the number parameter reference the same value

```
def main():
    value = 5
    show_double(value)


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



show_double ()

- As we do, remember that the number parameter variable will be assigned the value that was passed to it as an argument.
- In this program, that number is 5.
- Line 11 assigns the value of the expression number * 2 to a local variable named result.
- Because number references the value 5, this statement assigns 10 to result.
- Line 12 displays the result variable.
- The following statement shows how the show_double function can be called with a numeric literal passed as an argument:

```
show_double(50)
```

- This statement executes the show_double function, assigning 50 to the number parameter.
- The function will print 100.

Parameter Variable Scope

- A variable is visible only to statements inside the variable's 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

Program 5-8 (multiple_args.py)

```
1  # This program demonstrates a function that accepts
2  # two arguments.
3
4  def main():
5      print('The sum of 12 and 45 is')
6      show_sum(12, 45)
7
8  # The show_sum function accepts two arguments
9  # and displays their sum.
10 def show_sum(num1, num2):
11     result = num1 + num2
12     print(result)
13
14 # Call the main function.
15 main()
```

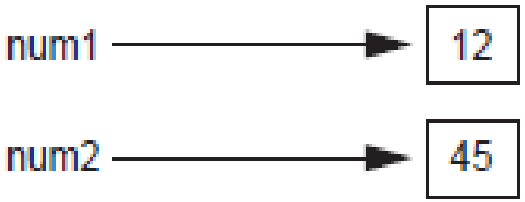
Program Output

```
The sum of 12 and 45 is
57
```

5-16 Two arguments passed to two parameters

```
def main():
    print('The sum of 12 and 45 is')
    show_sum(12, 45)
```

```
def show_sum(num1, num2):
    result = num1 + num2
    print(result)
```



Suppose we were to reverse the order in which the arguments are listed in the function call, as shown here:

```
show_sum(45, 12)
```

This would cause 45 to be passed to the num1 parameter and 12 to be passed to the num2

Passing strings

```
1  # This program demonstrates passing two string
2  # arguments to a function.
3
4  def main():
5      first_name = input('Enter your first name: ')
6      last_name = input('Enter your last name: ')
7      print('Your name reversed is')
8      reverse_name(first_name, last_name)
9
10 def reverse_name(first, last):
11     print(last, first)
12
13 # Call the main function.
14 main()
```

Program Output (with input shown in bold)

```
Enter your first name: Matt Enter
Enter your last name: Hoyle Enter
Your name reversed is
Hoyle Matt
```

Making Changes to Parameters

Program 5-10 *(continued)*

```
 4  def main():
 5      value = 99
 6      print('The value is', value)
 7      change_me(value)
 8      print('Back in main the value is', value)
 9
10  def change_me(arg):
11      print('I am changing the value.')
12      arg = 0
13      print('Now the value is', arg)
14
15  # Call the main function.
16  main()
```

Program Output

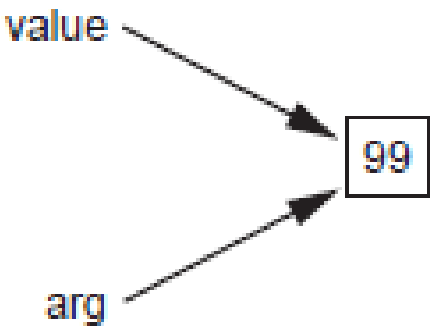
```
The value is 99
I am changing the value.
Now the value is 0
Back in main the value is 99
```

Making Changes to Parameters…contd

5-17 The value variable is passed to the change_me function

```
def main():
    value = 99
    print('The value is', value)
    change_me(value)
    print('Back in main the value is', value)


def change_me(arg):
    print('I am changing the value.')
    arg = 0
    print('Now the value is', arg)
```



- 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*.
- This is a way that one function can communicate with another function. The communication channel works in only one direction

Keyword Arguments

- 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*.

Keyword Arguments example

Program 5-12 (keyword_string_args.py)

```
1  # This program demonstrates passing two strings as
2  # keyword arguments to a function.
3
4  def main():
5      first_name = input('Enter your first name: ')
6      last_name = input('Enter your last name: ')
7      print('Your name reversed is')
8      reverse_name(last=last_name, first=first_name)
9
10 def reverse_name(first, last):
11     print(last, first)
12
13 # Call the main function.
14 main()
```

Program Output (with input shown in bold)

```
Enter your first name: Matt [Enter]
Enter your last name: Hoyle [Enter]
Your name reversed is
Hoyle Matt
```

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.
- Otherwise an error will occur.

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

- # This will cause an ERROR!

```
show_interest(1000.0, rate=0.01, 10)
```

Global Variables and Global Constants

- A global variable is accessible to all the functions in a program file.
- Variable is local to that function 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.

Example


```
1   # Create a global variable.
2   my_value = 10
3
4   # The show_value function prints
5   # the value of the global variable
6   def show_value():
7       print(my_value)
8
9   # Call the show_value function.
10  show_value()
```

Program Output

```
10
```

Modify global variables

Program 5-14 (global2.py)

```
1  # Create a global variable.
2  number = 0
3
4  def main():
5      global number
6      number = int(input('Enter a number: '))
7      show_number()
8
9  def show_number():
10     print('The number you entered is', number)
11
12 # Call the main function.
13 main()
```

Program Output

```
Enter a number: 55 Enter
The number you entered is 55
```

- Most programmers agree that you should restrict the use of global variables, or not use them at all.
- The reasons are as follows:

  - Global variables make debugging difficult. Any statement in a program file can change the value of a global variable. If you find that the wrong value is being stored in a global variable, you have to track down every statement that accesses it to determine where the bad value is coming from. In a program with thousands of lines of code, this can be difficult.
  - Functions that use global variables are usually dependent on those variables. If you want to use such a function in a different program, most likely you will have to redesign it so it does not rely on the global variable.
  - Global variables make a program hard to understand. A global variable can be modified by any statement in the program. If you are to understand any part of the program that uses a global variable, you have to be aware of all the other parts of the program that access the global variable.
- In most cases, you should create variables locally and pass them as arguments to the functions that need to access them.

Global Constants

- Although you should try to avoid the use of global variables, it is permissible to use global constants in a program.
- A *global constant* is a global name that references a value that cannot be changed.
- Because a global constant's value cannot be changed during the program's execution, you do not have to worry about many of the potential hazards that are associated with the use of global variables.
- Although the Python language does not allow you to create true global constants, you can simulate them with global variables.
- If you do not declare a global variable with the global key word inside a function, then you cannot change the variable's assignment inside that function.

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.**

Void function

- A void function is a group of statements that exist within a program for the purpose of performing a specific task.
- When you need the function to perform its task, you call the function.
- This causes the statements inside the function to execute.
- When the function is finished, control of the program returns to the statement appearing immediately after the function call.

Value-returning 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, as well as most other programming languages, comes with a *standard library* of functions that have already been written for you.
- These functions, known as *library functions*, make a programmer's job easier because they perform many of the tasks that programmers commonly need to perform.
- In fact, you have already used several of Python's library functions.
- Some of the functions that you have used are print, input, and range.
- Python has many other library functions.

Python's library functions

- Some of Python's library functions are built into the Python interpreter.
- If you want to use one of these built-in functions in a program, you simply call the function.
- This is the case with the print, input, range, and other functions that you have already learned about.
- Many of the functions in the standard library, however, are stored in files that are known as *modules*.
- These modules, which are copied to your computer when you install Python, help organize the standard library functions.
- For example, functions for performing math operations are stored together in a module, functions for working with files are stored together in another module, and so on.

import

- In order to call a function that is stored in a module, you have to write an import statement at the top of your program.
- An import statement tells the interpreter the name of the module that contains the function.
- For example, 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.

Black Boxes

- Because you do not see the internal workings of library functions, many programmers think of them as *black boxes*.
- The term “black box” is used to describe any mechanism that accepts input, performs some operation (that cannot be seen) using the input, and produces output.

Generating Random Numbers

- Random numbers are useful for lots of different programming tasks.
- The following are just a few examples.
  - Random numbers are commonly used in games. For example, computer games that let the player roll dice use random numbers to represent the values of the dice.
  - Programs that show cards being drawn from a shuffled deck use random numbers to represent the face values of the cards.
  - Random numbers are useful in simulation programs.
  - In some simulations, the computer must randomly decide how a person, animal, insect, or other living being will behave.
  - Formulas can be constructed in which a random number is used to determine various actions and events that take place in the program.
  - Random numbers are useful in statistical programs that must randomly select data for analysis.
  - Random numbers are commonly used in computer security to encrypt sensitive data.

import ....random

- 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

randint()

- The first random-number generating function that we will discuss is named randint.
- Because the randint function is in the random module, we will need to use dot notation to refer to it in our program.
- In dot notation, the function's name is random.randint.
- On the left side of the dot (period) is the name of the module, and on the right side of the dot is the name of the function.
- The following statement shows an example of how you might call the randint function.
- number = random.randint (1, 100)
- The part of the statement that reads random.randint(1, 100) is a call to the randint function.
- Notice that two arguments appear inside the parentheses: 1 and 100.
- These arguments tell the function to give an integer random number in the range of 1 through 100

Example

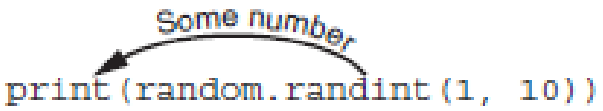
Program 5-16 (random_numbers.py)

```
1  # This program displays a random number
2  # in the range of 1 through 10.
3  import random
4
5  def main():
6      # Get a random number.
7      number = random.randint(1, 10)
8      # Display the number.
9      print('The number is', number)
10
11 # Call the main function.
12 main()
```

Program Output

```
The number is 7
```

igure 5-22 Displaying a random number



A random number in the range of
1 through 10 will be displayed.

Program 5-18 shows how you could simplify Program 5-17. This program also displays five random numbers, but this program does not use a variable to hold those numbers. The `randint` function's return value is sent directly to the `print` function in line 7.

Program 5-18 (random_numbers3.py)

```
1  # This program displays five random
2  # numbers in the range of 1 through 100.
3  import random
4
5  def main():
6      for count in range(5):
7          print(random.randint(1, 100))
8
9  # Call the main function.
10 main()
```

Program Output

89
7
16
41
12

```
1   # This program the rolling of dice.
2   import random
3
4   # Constants for the minimum and maximum random numbers
5   MIN = 1
6   MAX = 6
7
8   def main():
9       # Create a variable to control the loop.
0       again = 'y'
1
2       # Simulate rolling the dice.
3       while again == 'y' or again == 'Y':
4           print('Rolling the dice ...')
5           print('Their values are:')
6           print(random.randint(MIN, MAX))
7           print(random.randint(MIN, MAX))
8
9           # Do another roll of the dice?
0           again = input('Roll them again? (y = yes): ')
1
2   # Call the main function.
3   main()
```

Program Output (with input shown in bold)

```
Rolling the dice ...
Their values are:
3
1
Roll them again? (y = yes): y Enter
Rolling the dice ...
Their values are:
1
1
Roll them again? (y = yes): y Enter
Rolling the dice ...
Their values are:
5
6
Roll them again? (y = yes): y Enter
```

Examples


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

- In this statement, a random number in the range of 1 through 10 is generated and then multiplied by 2.
- The result is a random even integer from 2 to 20 assigned to the x variable.

The randrange, random, and uniform Functions

- The randrange function takes the same arguments as the range function.
- The difference is that the randrange function does not return a list of values.
- Instead, it returns a randomly selected value from a sequence of values.

Usage

- For 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.

Usage...contd

- 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]

random()

- Both the randint and the randrange functions return an integer number.
- The random function, however, 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).
- Here is an example: number = random.random()
- The uniform function also returns a random floating-point number, but allows you to specify the range of values to select from.
- Here is an 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.
- Although we commonly refer to them as random numbers, 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.

Seed() contd..

- If the same seed value were always used, the random number functions would always generate the same series of pseudorandom numbers.
- Because the system time changes every hundredth of a second, it is a fairly safe bet that each time you import the random module, a different sequence of random numbers will be generated.
- However, there may be some applications in which you want to always generate the same sequence of random numbers.

Seed() ....contd

- In this example, 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.
- To demonstrate, look at the following interactive sessions.

```
>>> import random
>>> random.seed(10)
>>> random.randint(1, 100)
58
>>> random.randint(1, 100)
43
>>> random.randint(1, 100)
58
>>> random.randint(1, 100)
21
```

Examples.....random

In [13]:
```
import random
print(random.randint(1, 10))
```

```
6
```

In [14]:
```
print(random.randint(1, 10))
```

```
5
```

In [24]:
```
number = random.randrange(10)

print (number)
```

```
1
```

In [25]:
```
print(random.randrange(10))
```

```
7
```

In [29]:
```
number = random.randrange(5,10)
print (number)

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

```
5
50
```

In [30]:
```
#randrange function returns a randomly selected value from the following sequence of numbers
# [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
number = random.randrange(0, 101, 10)
print(number)
```

```
10
```

In [32]:
```
number = random.randrange(0, 101, 3)
print(number)
```

```
87
```

In [33]:
```
#returns a random floating point number in the range of 0.0 up to 1.0 (but not including 1.0).
number = random.random()
print(number)
#returns a random floating-point number in the specified range of values
number = random.uniform(1.0, 10.0)
print(number)
```

```
0.05610257476796077
2.9774576841848495
```

In [43]:
```
import random
random.seed(10)
x=random.randint(1, 100)
y=random.randint(1, 100)
print(x,y)
```

```
74 5
```

In [44]:
```
random.seed(10)
x=random.randint(1, 100)
y=random.randint(1, 100)
print(x,y)
```

```
74 5
```

In [45]:
```
random.seed(10)
x=random.randint(1, 100)
y=random.randint(1, 100)
print(x,y)
```

```
74 5
```

Writing Your Own Value-Returning Functions

- **Value-returning function has a return statement that returns a value back to the part of the program that called it.**
- You write a value-returning function in the same way that you write a void function, with one exception: a value-returning function must have a return statement.
- Here is the general format of a value-returning function definition in Python:

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

- One of the statements in the function must be a return statement, which takes the following form:

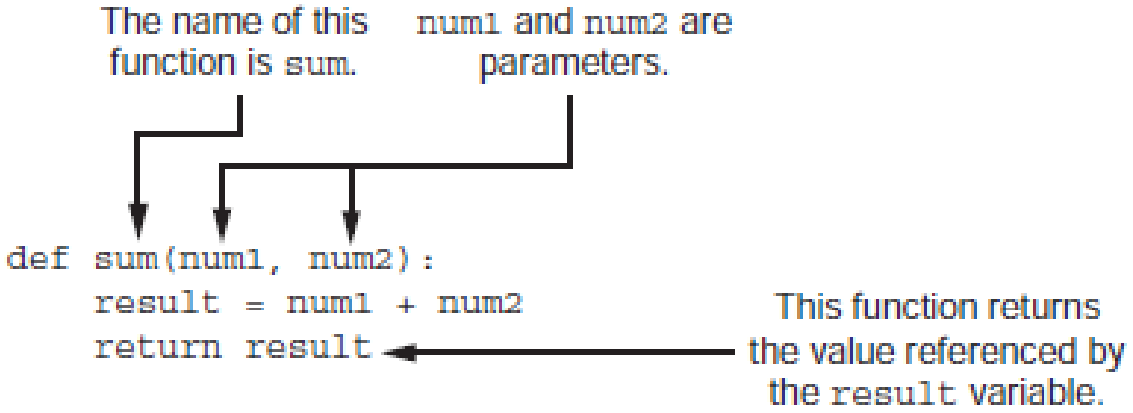
```
return expression
```

Return


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

Figure 5-23 illustrates various parts of the function.

5-23 Parts of the function



Example

```
1  # This program uses the return value of a function.
2
3  def main():
4      # Get the user's age.
5      first_age = int(input('Enter your age: '))
6
7      # Get the user's best friend's age.
8      second_age = int(input("Enter your best friend's age: "))
9
10     # Get the sum of both ages.
11     total = sum(first_age, second_age)
12
13     # Display the total age.
14     print('Together you are', total, 'years old.')
15
16 # The sum function accepts two numeric arguments and
17 # returns the sum of those arguments.
18 def sum(num1, num2):
19     result = num1 + num2
20     return result
21
22 # Call the main function.
23 main()
```

Program Output (with input shown in bold)

```
Enter your age: 22 Enter
Enter your best friend's age: 24 Enter
Together you are 46 years old.
```

Returning Strings & Boolean variables


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

```
def is_even(number):
    # Determine whether number is even. If it is,
    # set status to true. Otherwise, set status
    # to false.
    if (number % 2) == 0:
        status = True
    else:
        status = False
    # Return the value of the status variable.
    return status
```

Returning Multiple Values

- return *expression1, expression2, etc.*

```
def get_name():
    # Get the user's first and last names.
    first = input('Enter your first name: ')
    last = input('Enter your last name: ')

    # Return both names.
    return first, last
```

When you call this function in an assignment statement, you need to use two variables on the left side of the = operator. Here is an example:

```
first_name, last_name = get_name()
```

- Note that the number of variables on the left side of the = operator must match the number of values returned by the function. Otherwise an error will occur

The math Module

- The Python standard library's math module contains numerous functions that can be used in mathematical calculations.
- The math module in the Python standard library contains several functions that are useful for performing mathematical operations.
- These functions typically accept one or more values as arguments, perform a mathematical operation using the arguments, and return the result.

Functions in math module

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.

Examples


```
# This program demonstrates the sqrt function.
import math

def main():
# Get a number.
number = float(input('Enter a number: '))

# Get the square root of the number.
square_root = math.sqrt(number)

# Display the square root.
print('The square root of', number, 'is',
    square_root)

# Call the main function.
main()
```

Program Output (with input shown in bold)

Enter a number: **25**
The square root of 25.0 is 5.0

```
# This program calculates the length of a right
# triangle's hypotenuse.
import math

def main():
# Get the length of the triangle's two sides.
a = float(input('Enter the length of side A: '))
b = float(input('Enter the length of side B: '))

# Calculate the length of the hypotenuse.
c = math.hypot(a, b)

# Display the length of the hypotenuse.
print('The length of the hypotenuse is', c)
# Call the main function.
main()
```

Program Output (with input shown in bold)

Enter the length of side A: **5.0**
Enter the length of side B: **12.0**
The length of the hypotenuse is 13.0

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
```

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.
- As your programs become larger and more complex, the need to organize your code becomes greater.
- You have already learned that a large and complex program should be divided into functions that each performs a specific task.
- As you write more and more functions in a program, you should consider organizing the functions by storing them in modules

Storing Functions in Modules

- Modules also make it 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.

Naming a module

- Notice that both of these files contain function definitions, but they do not contain code that calls the functions.
- That will be done by the program or programs that import these modules.
- Before continuing, we should mention the following things about module names:
  - A module's file name should end in .py.
  - If the module's file name does not end in .py you will not be able to import it into other programs.
- A module's name cannot be the same as a Python key word.
- An error would occur, for example, if you named a module for.

Example


```
# The circle module has functions that perform
# calculations related to circles.
import math

# The area function accepts a circle's radius as an
# argument and returns the area of the circle.
def area(radius):
return math.pi * radius**2
# The circumference function accepts a circle's
# radius and returns the circle's circumference.
def circumference(radius):
return 2 * math.pi * radius
```

```
# The rectangle module has functions that perform
# calculations related to rectangles.

# The area function accepts a rectangle's width and
# length as arguments and returns the rectangle's area.
def area(width, length):
return width * length

# The perimeter function accepts a rectangle's width
# and length as arguments and returns the rectangle's
# perimeter.
def perimeter(width, length):
return 2 * (width + length)
```

Importing user defined modules

- To use these modules in a program, you import them with the import statement
- When the Python interpreter reads this statement it will look for the file circle.py in the same folder as the program that is trying to import it.
- If it finds the file, it will load it into memory. If it does not find the file, an error occurs.
- Once a module is imported you can call its functions.
- Assuming that radius is a variable that is assigned the radius of a circle, here is an example of how we would call the area and circumference functions:

```
my_area = circle.area(radius)
my_circum = circle.circumference(radius)
```

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*.

```
# The display_menu function displays a menu.
def display_menu():
print(' MENU')
print('1) Area of a circle')
print('2) Circumference of a circle')
print('3) Area of a rectangle')
print('4) Perimeter of a rectangle')
print('5) Quit')
```

Time module

- A Python program can handle date and time in several ways.
- Converting between date formats is a common chore for computers. Python's time and calendar modules help track dates and times.
- **What is Tick?**
- Time intervals are floating-point numbers in units of seconds.
- Particular instants in time are expressed in seconds since 12:00am, January 1, 1970*epoch*.
- There is a popular **time module available in Python which provides functions for working with** times, and for converting between representations.
- The function *time.time() returns the current* system time in ticks since 12:00am, January 1, 1970*epoch*.

Example

- import time; # This is required to include time module.
- ticks = time.time()
- print "Number of ticks since 12:00am , January 1, 1970:", ticks

Time Tuple

Index	Field	Values
0	4-digit year	2008
1	Month	1 to 12
2	Day	1 to 31
3	Hour	0 to 23
4	Minute	0 to 59
5	Second	0 to 61 (60 or 61 are leap-seconds)
6	Day of Week	0 to 6 (0 is Monday)
7	Day of year	1 to 366 (Julian day)
8	Daylight savings	-1, 0, 1, -1 means library determines DST

Getting current time

- To translate a time instant from a *seconds since the epoch* floating-point value into a time-tuple, pass the floating-point value to a function (e.g., localtime) that returns a time-tuple with all nine items valid.

```
import time;

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

This would produce the following result, which could be formatted in any other presentable form –

```
Local current time : time.struct_time(tm_year=2013, tm_mon=7,
tm_mday=17, tm_hour=21, tm_min=26, tm_sec=3, tm_wday=2, tm_yday=198, tm_isdst=0)
```

Getting formatted time

- You can format any time as per your requirement, but simple method to get time in readable format is asctime()

```
import time;

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

This would produce the following result –

```
Local current time : Tue Jan 13 10:17:09 2009
```

Getting calendar for a month

- The calendar module gives a wide range of methods to play with yearly and monthly calendars. Here, we print a calendar for a given month (Jan 2008)

```
import calendar

cal = calendar.month(2008, 1)
print "Here is the calendar:"
print cal
```

This would produce the following result –

```
Here is the calendar:
   January 2008
Mo Tu We Th Fr Sa Su
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

The *time* Module

- time.altzone: The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC *asinWesternEurope, includingtheUK. Only use* this if daylight is nonzero.
- time.asctime[*tupletime]:* Accepts a time-tuple and returns a readable 24-character string such as 'Tue Dec 11 18:07:14 2008'.
- time.clock: Returns the current CPU time as a floating-point number of seconds. To measure computational costs of different approaches, the value of time.clock is more useful than that of time.time.
- time.ctime[*secs]:* Like asctime*localtime(secs) and without arguments is like asctime*
- time.gmtime[*secs]:* Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the UTC time. Note : t.tm_isdst is always 0
- time.localtime[*secs]:* Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the local time *t. tmisdstis0or1, depending on whether DST applies to instant secs by local rules.*
- time.mktime*tupletime:* Accepts an instant expressed as a time-tuple in local time and returns a floating-point value with the instant expressed in seconds since the epoch.

The *time* Module

- time.sleep*secs:* Suspends the calling thread for secs seconds.
- time.strftime*fmt[, tupletime]:* Accepts an instant expressed as a time-tuple in local time and returns a string representing the instant as specified by string fmt.
- time.strptime*str, fmt* =Parses str according to format string fmt and returns the instant in time-tuple format.
- time.time: Returns the current time instant, a floating-point number of seconds since the epoch.
- time.tzset: Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.

The *time* Module

time.timezone

Attribute time.timezone is the offset in seconds of the local time zone *withoutDST* from UTC
> 0*intheAmericas*; <= 0*inmostofEurope*, *Asia*, *Africa*.

time.tzname

Attribute time.tzname is a pair of locale-dependent strings, which are the names of the local time zone without and with DST, respectively.

The *calendar Module*

- The calendar module supplies calendar-related functions, including functions to print a text calendar for a given month or year.
- By default, calendar takes Monday as the first day of the week and Sunday as the last one.
- To change this, call calendar.setfirstweekday function.

References

- Tony Gaddis, “Starting out with Python”, 3rd Edition, Global Edition, Pearson Education
- https://www.tutorialspoint.com/python/python_date_time.htm