Object Oriented Programming II

Dr. Burak Kaleci

February 22, 2019

Content

Functions

Defining and Calling a Void Function

Value-Returning Functions

The math Module

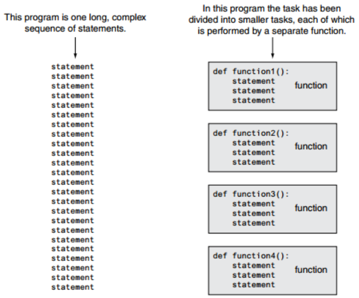
Storing Functions in Modules

Examples

Introduction

- 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.

Introduction


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.
function
function
function
function

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

```
def function1():
    statement
    statement
    statement

def function2():
    statement
    statement
    statement

def function3():
    statement
    statement
    statement

def function4():
    statement
    statement
    statement
```

- This approach is sometimes called **divide and conquer** because a large task is divided into several smaller tasks that are easily performed.
- Figure illustrates this idea by comparing two programs:
  - one that uses a long complex sequence of statements to perform a task
  - Another that divides a task into smaller tasks, each of which is performed by a separate function.
- When using functions in a program, you generally isolate each task within the program in its own function.

Introduction

- A program that has been written with each task in its own function is called a **modularized program**.
- Benefits of Modularizing a Program with Functions:
  - **Simpler Code**
  - **Code Reuse**
  - **Better Testing**
  - **Faster Development**
  - **Easier Facilitation of Teamwork**

Void Functions and Value-Returning Functions

- In Python, there are two types of functions:
  - **void functions**
  - **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, then returns a value back to the statement that called it.
- 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.

Function Names

- Python requires that you follow the same rules that you follow when naming variables, which we recap here:
  - 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.

A function's name should be descriptive enough so anyone reading your code can reasonably guess what the function does.

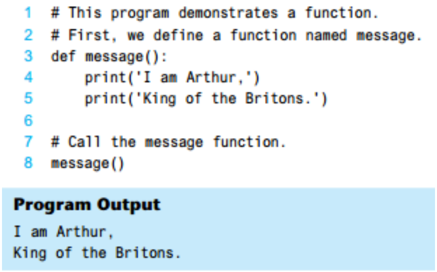
Defining and Calling a Function

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

```
def function_name():
    statement
    statement
    statement
```

- 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.

Defining a Function



```
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.
```

- Lines 3-5 define 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

```
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

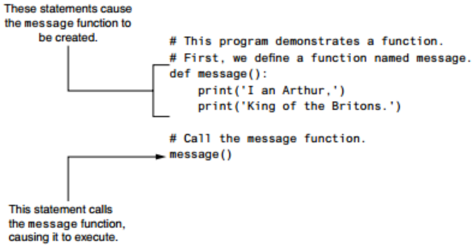
```
I am Arthur,
King of the Britons.
```

- 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 (Line 8).
- **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.

Defining and Calling a Function

- In Line 3, interpreter reads the def statement.
- **This causes a function named message to be created in memory, containing the block of statements in lines 4 and 5.**
- The interpreter executes the statement in line 8, which is a function call. This causes the message function to execute, which prints the two lines of output.


These statements cause the message function to be created.
This statement calls the message function, causing it to execute.

```
# This program demonstrates a function.
# First, we define a function named message.
def message():
    print('I an Arthur,')
    print('King of the Britons.')

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

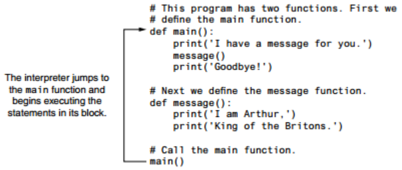
Two Functions

- The previous program has only one function, but 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.

Two Functions

```
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()
```

- The definition of the main function appears in lines 3 through 6, and the definition of the message function appears in lines 9 through 11.


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

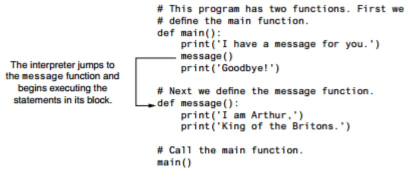
```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

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

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

- The statement in line 14 calls the main function.
- The first statement in the main function calls the print function in line 4.
- It displays the string 'I have a message for you'.

Two Functions

```
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()
```

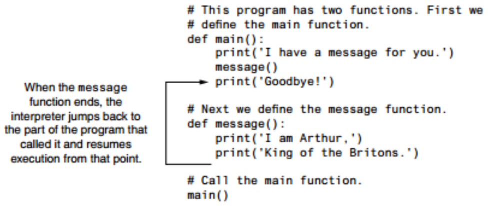
- Then, the statement in line 5 calls the message function.
- This causes the interpreter to jump to the message function.


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()
```

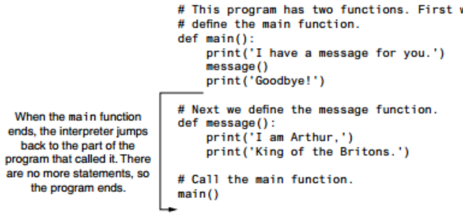
- After the statements in the message function have executed, the interpreter returns to the main function and resumes with the statement that immediately follows the function call.
- There are no more statements to execute, so the program ends.

Two Functions



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()
```



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.

```
# This program has two functions. First w
# 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()
```

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 belongs to another function.**

Local Variables

```
 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.

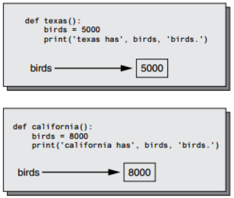
Scope and Local Variables

```
# This program demonstrates two functions that
# have local variables with the same name.

def main():
    # Call the texas function.
    texas()
    # Call the california function.
    california()

# Definition of the texas function. It creates
# a local variable named birds.
def texas():
    birds = 5000
    print('texas has', birds, 'birds.')

# Definition of the california function. It also
# creates a local variable named birds.
def california():
    birds = 8000
    print('california has', birds, 'birds.')

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

- 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.
- Because a function's local variables are hidden from other functions, the other functions may have their own local variables with the same name.
- In addition to the main function, this program has two other functions: texas and california.

Scope and Local Variables

- **Although there are two separate variables named birds in this program, only one of them is visible at a time because they are in different functions.**
- When the texas function is executing, the birds variable that is created in line 13 is visible.
- When the california function is executing, the birds variable that is created in line 19 is visible.



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

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

Passing Arguments to Functions

- Sometimes it is useful not only to call a function, but also to send one or more pieces of data into the function.
- Pieces of data that are sent into a function are known as **arguments**.
- The function can use its arguments in calculations or other operations.
- If you want a function to receive arguments when it is called, you must equip the function with one or more parameter variables.
- A parameter variable,often simply called a **parameter**,is a special variable that is assigned the value of an argument when a function is called.

Passing Arguments to Functions

```
# This program demonstrates an argument being
# passed to a function.

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

# The show_double function accepts an argument
# and displays double its value.
def show_double(number):
    result = number * 2
    print(result)

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

- The purpose of show_double function is to accept a number as an argument and display the value of that number doubled.
- **Look at the function header and notice the word number that appear 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.

Passing Arguments to Functions

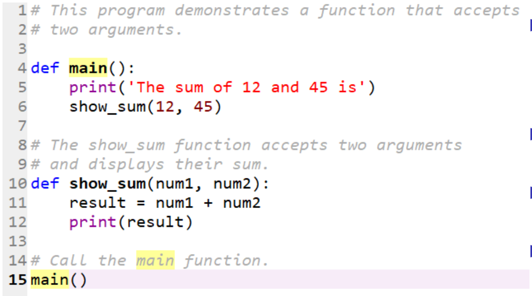
```
1   # This program demonstrates an argument being
2   # passed to a function.
3
4   def main():
5       value = 5
6       show_double(value)
7
8   # The show_double function accepts an argument
9   # and displays double its value.
10  def show_double(number):
11      result = number * 2
12      print(result)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

- When this program runs, the main function is called in line 15.
- Inside the main function, line 5 creates a local variable named value, assigned the value 5.
- Line 6 calls the show_double function.
- Notice value appears inside the parentheses.
- This means that value is being passed as an argument to the show_double function.
- When this statement executes, the show_double function will be called, and the number parameter will be assigned the same value as the value variable.

Parameter Variable Scope

- Remember that a variable's scope is the part of the program in which the variable may be accessed.
- A variable is visible only to statements inside the variables 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.

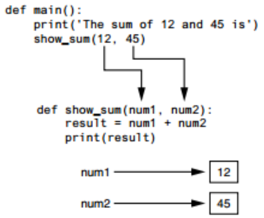
Passing Multiple Arguments



```
 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()
```

- Often it's useful to write functions that can accept multiple arguments.
- The program shows a function named show_sum, that accepts two arguments.
- Notice two parameter variable names, num1 and num2, appear inside the parentheses in the show_sum function header.
- **This is often referred to as a parameter list.**
- Also notice a comma separates the variable names.
- The statement in line 6 calls the show_sum function and passes two arguments: 12 and 45.
- These arguments are passed by position to the corresponding parameter variables in the function.

Passing Multiple Arguments



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

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

- In other words, the first argument is passed to the first parameter variable, and the second argument is passed to the second parameter variable.
- So, this statement causes 12 to be assigned to the num1 parameter and 45 to be assigned to the num2 parameter.
- Suppose we were to reverse the order in which the arguments are listed in the function call.
- This would cause 45 to be passed to the num1 parameter, and 12 to be passed to the num2 parameter.
- The following code shows another example. This time, we are passing variables as arguments.

  value1=2
  value2=3
  show_sum(value1,value2)
- When the show_sum function executes as a result of this code, the num1 parameter will be assigned the value 2, and the num2 parameter will be assigned the value 3.

Passing Multiple Arguments

```
# This program demonstrates passing two string
# arguments to a function.

def main():
    first_name = input('Enter your first name: ')
    last_name = input('Enter your last name: ')
    print('Your name reversed is')
    reverse_name(first_name, last_name)

def reverse_name(first, last):
    print(last, first)

# Call the main function.
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

```
 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
```

- When an argument is passed to a function in Python, the function parameter variable will reference the argument's value.
- **However, any changes that are made to the parameter variable will not affect the argument.**
- The main function creates a local variable named value in line 5, assigned the value 99.
- The statement in line 6 displays 'The value is 99'. The value variable is then passed as an argument to the change_me function in line 7.

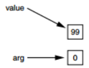
Making Changes to Parameters

```
 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
```

- This means that in the change_me function, the arg parameter will also reference the value 99.
- Inside the change_me function, in line 12, the arg parameter is assigned the value 0.
- This reassignment changes arg, but it does not affect the value variable in main.
- The two variables now reference different values in memory



Making Changes to Parameters

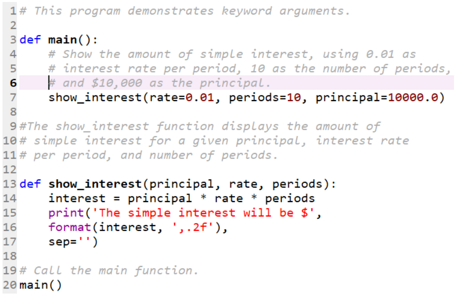
- The form of argument passing that is used in Python, where a function cannot change the value of an argument that was passed to it, is commonly called **pass by value**.
- This is a way that one function can communicate with another function.
- The communication channel works in only one direction, however.
- The calling function can communicate with the called function, but the called function cannot use the argument to communicate with the calling function.
- Later, you will learn how to write a function that can communicate with the part of the program that called it **by returning a value**.

Keyword Arguments

- The previous programs demonstrate how arguments are passed by position to parameter variables in a function.
- Most programming languages match function arguments and parameters this way.
- 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



```
 1 # This program demonstrates keyword arguments.
 2 
 3 def main():
 4     # Show the amount of simple interest, using 0.01 as
 5     # interest rate per period, 10 as the number of periods,
 6     # and $10,000 as the principal.
 7     show_interest(rate=0.01, periods=10, principal=10000.0)
 8 
 9 #The show_interest function displays the amount of
10 # simple interest for a given principal, interest rate
11 # per period, and number of periods.
12 
13 def show_interest(principal, rate, periods):
14     interest = principal * rate * periods
15     print('The simple interest will be $',
16     format(interest, ',.2f'),
17     sep='')
18 
19 # Call the main function.
20 main()
```

- This program uses a function named show_interest that displays the amount of simple interest earned by a bank account for a number of periods.

- The function accepts the arguments principal(for the account principal), rate (for the interest rate per period), and periods(for the number of periods).
- When the function is called in line 7, the arguments are passed as keyword arguments.
- Notice in line 7 the order of the keyword arguments does not match the order of the parameters in the function header in line 13. Because a keyword argument specifies which parameter the argument should be passed into, its position in the function call does not matter.

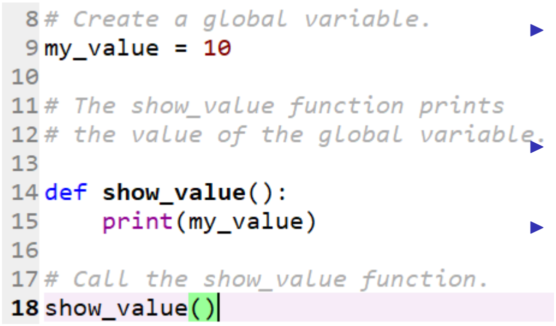
Mixing Keyword Arguments with Positional Arguments

- 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.**
- Here is an example of how we might call the show_interest function using both positional and keyword arguments:
  ```
  show_interest(10000.0, rate=0.01, periods=10)
  ```
- 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.
- The following function call will cause an error, however, because a non-keyword argument follows a keyword argument:
  ```
  show_interest(1000.0, rate=0.01, 10)
  ```

Global Variables

- You've learned that 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.

Global Variables



```
 8 # Create a global variable.
 9 my_value = 10
10
11 # The show_value function prints
12 # the value of the global variable.
13
14 def show_value():
15     print(my_value)
16
17 # Call the show_value function.
18 show_value()
```

- The assignment statement in line 2 creates a variable named my_value.
- Because this statement is outside any function, it is global.
- When the show_value function executes, the statement in line 7 prints the value referenced by my_value.

Global Variables

```
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
```

- An additional step is required if you want a statement in a function to assign a value to a global variable.
- In the function, you must declare the global variable.
- The assignment statement in line 2 creates a global variable named number.
- Notice inside the main function, line 5 uses the global key word to declare the number variable.
- This statement tells the interpreter that the main function intends to assign a value to the global number variable.
- That's just what happens in line 6. The value entered by the user is assigned to number.

Global Constants

- 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

- 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, as well as most 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.
- **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 about which you have already learned.

Standard Library Functions and the import Statement

- 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.
- 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.

Standard Library Functions and the import Statement

- 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.**

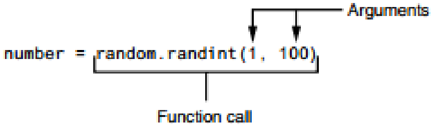
Generating Random Numbers

- Random numbers are useful for lots of different programming tasks.
- 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:**

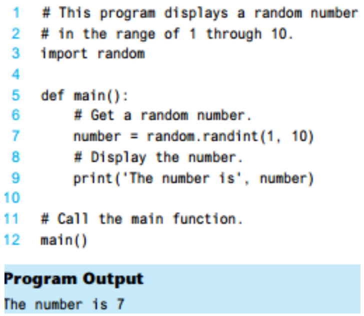
  ```
  import random
  ```
- The first random-number generating function that we will discuss is named **randint**.
- Because the **randint** function is in the **random module**, we will **need to use dot notation to refer to it** in our program.
- In dot notation, the function's name is random.randint.
- 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**.

Generating Random Numbers

- 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 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. (The values 1 and 100 are included in the range.)
- Notice the call to the randint function appears on the right side of an =operator.
- When the function is called, it will generate a random number in the range of 1 through 100 then return that number.
- The number that is returned will be assigned to the number variable.


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

An Example



```
 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
```

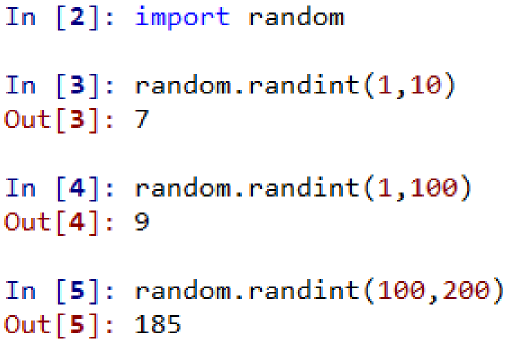
- The statement in line 7 generates a random number in the range of 1 through 10 and assigns it to the number variable.
- The program output shows that the number 7 was generated, but this value is arbitrary. If this were an actual program, it could display any number from 1 to 10.

Another Example

```
# This program displays five random
# numbers in the range of 1 through 100.
import random

def main():
    for count in range(5):
        print(random.randint(1, 100))

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

- If you just want to display a random number, it is not necessary to assign the random number to a variable.
- You can send the random function's return value directly to the print function, as shown here:
  print(random.randint(1, 10))
- When this statement executes, the randint function is called. The function generates a random number in the range of 1 through 10.
- That value is returned and sent to the print function. As a result, a random number in the range of 1 through 10 will be displayed.

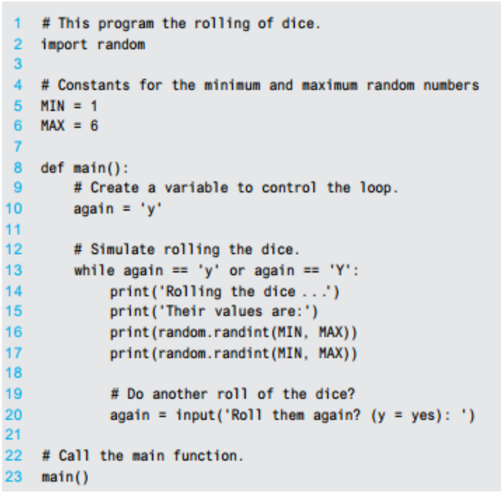
Experimenting with Random Numbers in Interactive Mode



```
In [2]: import random

In [3]: random.randint(1,10)
Out[3]: 7

In [4]: random.randint(1,100)
Out[4]: 9

In [5]: random.randint(100,200)
Out[5]: 185
```

- To get a feel for the way the randint function works with different arguments, you might want to experiment with it in interactive mode.
- The statement in line 2 imports the random module. **(You have to write the appropriate importstatements in interactive mode, too.)**
- The statement in line 3 calls the randint function, passing 1 and 10 as arguments.

Dice example



```
# This program the rolling of dice.
import random

# Constants for the minimum and maximum random numbers
MIN = 1
MAX = 6

def main():
    # Create a variable to control the loop.
    again = 'y'

    # Simulate rolling the dice.
    while again == 'y' or again == 'Y':
        print('Rolling the dice ...')
        print('Their values are:')
        print(random.randint(MIN, MAX))
        print(random.randint(MIN, MAX))

        # Do another roll of the dice?
        again = input('Roll them again? (y = yes): ')

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

The randrange, random, and uniform Functions

- The standard library's random module contains numerous functions for working with random numbers.
- In addition to the **randint** function, you might find the **randrange**, **random**, and **uniform** functions useful.
- 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.
- 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.

The randrange, random, and uniform Functions

- 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 randrangefunction returns a randomly selected value from the following sequence of numbers:
  [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

The randrange, random, and uniform Functions

- 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)**
- 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.**
- The system time is an integer that represents the current date and time, down to a hundredth of a second.

Random Number Seeds

- 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.
- If that is the case, you can call the **random.seed** function to specify a seed value. Here is an example:

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

Writing Your Own Value-Returning Functions

- 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
```

- 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

Writing Your Own Value-Returning Functions

- Here is a simple example of a value-returning function:

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

- The purpose of this function is to accept two integer values as arguments and return their sum.
- The first statement in the function's block assigns the value of num1+ num2 to the result variable.
- **Next, the return statement executes, which causes the function to end execution and sends the value referenced by the result variable back to the part of the program that called the function.**

Writing Your Own Value-Returning Functions

```
 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()
```

- In the main function, the program gets two values from the user and stores them in the first_age and second_age variables.
- The statement in line 11 calls the sum function, passing first_age and second_age as arguments.
- The value that is returned from the sum function is assigned to the total variable.

Returning Strings

- So far, you've seen examples of functions that return numbers.
- You can also write functions that return strings.
- For example, the following function prompts the user to enter his or her name, then returns the string that the user entered:

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

Returning Boolean Values

- Python allows you to write Boolean functions,which return either True or False.
- You can use a Boolean function to test a condition, then return either True or False to indicate whether the condition exists.
- Boolean functions are useful for simplifying complex conditions that are tested in decision and repetition structures.
- For example, you could write a Boolean function named is_even that accepts a number as an argument and returns True if the number is even, or False otherwise.
- Not only is this logic easier to understand, but now you have a function that you can call in the program any time you need to test a number to determine whether it is even.
- You can also use Boolean functions to simplify complex input validation code.

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.
- As an example, look at the following definition for a function named get_name.
- The function prompts the user to enter his or her first and last names. These names are stored in two local variables: first and last.
- The return statement returns both of the variables.

Returning Multiple Values

```
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()
- The values listed in the return statement are assigned, in the order that they appear, to the variables on the left side of the =operator.
- After this statement executes, the value of the first variable will be assigned to first_name, and the value of the last variable will be assigned to last_name.
- Note 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.

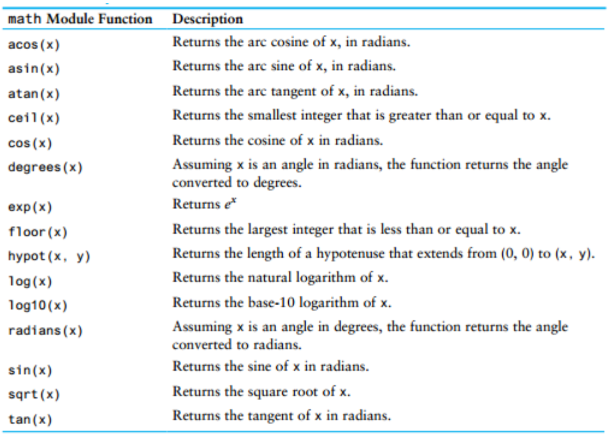
Introduction

- 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.
- For example, one of the functions is named **sqrt**. The sqrt function accepts an argument and returns the square root of the argument.
- Here is an example of how it is used:
  ```
  result = math.sqrt(16)
  ```
- This statement calls the sqrtfunction, passing 16 as an argument. The function returns the square root of 16, which is then assigned to the result variable.

An Example

```
# 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()
```

math module functions



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.

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

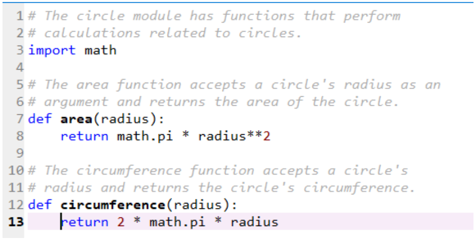
Introduction

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

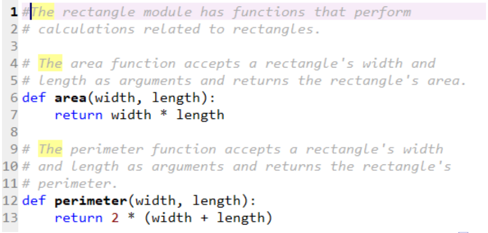
Introduction

- Let's look at a simple example. Suppose your instructor has asked you to write a program that calculates the following:
  - The area of a circle
  - The circumference of a circle
  - The area of a rectangle
  - The perimeter of a rectangle
- There are obviously two categories of calculations required in this program: those related to circles, and those related to rectangles.
- You could write all of the circle-related functions in one module, and the rectangle-related functions in another module.

Circle and Rectangle Modules



```python
# 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
```



```python
#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)
```

Circle and Rectangle Modules

- Notice 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**.
- **To use these modules in a program, you import them with the import statement.**
- Here is an example of how we would import the circle module:

  import circle

Circle and Rectangle Modules

- 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 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)
```

Example 1

Write a Python program to to convert Spherical coordinates (r,θ,ϕ) to Cartesian coordinates (x,y,z).

- Enter the (r,θ,ϕ) values from the keyboard and print them.
- Then, use **convert function** to calculate and print the cartesian coordinates (x,y,z) according to the following formulas:

$$x = r\sin(\theta)\cos(\phi)$$
$$y = r\sin(\theta)\sin(\phi)$$
$$z = r\cos(\theta)$$

- Test your program with r=10, $\theta = 60^o$, and $\phi = 90^o$.

Example 2

Consider we have a second-order system with damping ratio ζ and natural frequency w_n. Write a C program to calculate w_d and t_r.

- Enter the ζ and w_n values from the keyboard and print them.
- Use **calculate function** to calculate w_d and t_r by using following formulas and print them.

$$w_d = w_n\sqrt{(1-\zeta^2)}$$
$$t_r = (\pi - tan^{-1}(\frac{w_d}{\zeta w_n}))/w_d$$

- Test your program with $w_n = 5$ and $\zeta = 0.6$.