

Chapter 2 Input, Processing, and Output

Displaying Screen Output

The `print` statement in Python displays output on the screen. Here is an example:

```
print 'Hello world'
```

The purpose of this statement is to display the message *Hello world* on the screen. Notice that after the word `print`, we have written `Hello world` inside single-quote marks. The quote marks will not be displayed when the statement executes. They simply mark the beginning and the end of the text that we wish to display.

Suppose your instructor tells you to write a program that displays your name and address on the computer screen. Program 2-1 shows an example of such a program, with the output that it will produce when it runs. (The line numbers that appear in a program listing in this book are *not* part of the program. We use the line numbers in our discussion to refer to parts of the program.)

Program 2-1 (*output.py*)

```
1  print 'Kate Austen'  
2  print '123 Dharma Lane'  
3  print 'Asheville, NC 28899'
```

This program is the Python version of **Program 2-1** in your textbook!

Program Output

```
Kate Austen  
123 Dharma Lane  
Asheville, NC 28899
```

Remember, these line numbers are **NOT** part of the program! Don't type the line numbers when you are entering program code. All of the programs in this booklet will show line numbers for reference purposes only.

It is important to understand that the statements in this program execute in the order that they appear, from the top of the program to the bottom. When you run this program, the first statement will execute, followed by the second statement, and followed by the third statement.

Strings and String Literals

In Python code, string literals must be enclosed in quote marks. As mentioned earlier, the quote marks simply mark where the string data begins and ends.

In Python you can enclose string literals in a set of single-quote marks (`'`) or a set of double-quote marks (`"`). The string literals in Program 2-1 are enclosed in single-quote marks, but the program could also be written as shown here:

```
print "Kate Austen"  
print "123 Dharma Lane"  
print "Asheville, NC 28899"
```

If you want a string literal to contain either a single-quote or an apostrophe as part of the string, you can enclose the string literal in double-quote marks. For example, Program 2-2 prints two strings that contain apostrophes.

Program 2-2 (*apostrophe.py*)

```
1 print "Don't fear!"  
2 print "I'm here!"
```

Program Output

```
Don't fear!  
I'm here!
```

Likewise, you can use single-quote marks to enclose a string literal that contains double-quotes as part of the string. Program 2-3 shows an example.

Program 2-3 (*display_quote.py*)

```
1 print 'Your assignment is to read "Hamlet" by tomorrow.'
```

Program Output

```
Your assignment is to read "Hamlet" by tomorrow.
```

Python also allows you to enclose string literals in triple quotes (either `"""` or `'''`). Triple quoted strings can contain both single quotes and double quotes as part of the string. The following statement shows an example:

```
print """I'm reading "Hamlet" tonight."""
```

This statement will print:

```
I'm reading "Hamlet" tonight.
```

Triple quotes can also be used to surround multiline strings, which is something that single and double quotes cannot be used for. Here is an example:

```
print """One  
Two  
Three"""
```

This statement will print:

```
One  
Two  
Three
```

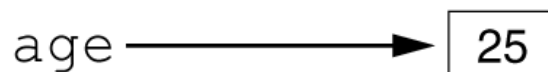
Variables

Variables are not declared in Python. Instead, you use an *assignment statement* to create a variable. Here is an example of an assignment statement:

```
age = 25
```

After this statement executes, a variable named `age` will be created and it will reference the value 25. This concept is shown in Figure 2-1. In the figure, think of the value 25 as being stored somewhere in the computer's memory. The arrow that points from `age` to the value 25 indicates that the name `age` references the value.

Figure 2-1 The `age` variable references the value 25



An assignment statement is written in the following general format:

variable = expression

The equal sign (=) is known as the *assignment operator*. In the general format, *variable* is the name of a variable and *expression* is a value, or any piece of code that results in a value. After an assignment statement executes, the variable listed on the left side of the = operator will reference the value given on the right side of the = operator.

The code in Program 2-4 demonstrates a variable. Line 1 creates a variable named `room` and assigns it the value 503. The `print` statements in lines 2 and 3 display a message. Notice that line 3 displays the value that is referenced by the `room` variable.

Program 2-4 (variable_demo.py)

```
1 room = 503
2 print 'I am staying in room number'
3 print room
```

Program Output

```
I am staying in room number
503
```

Variable Naming Rules

You may choose your own variable names in Python, as long as you do not use any of the Python key words. The key words make up the core of the language and each has a specific purpose. Table 2-1 shows a list of the Python key words.

Additionally, you must follow these rules when naming variables in Python:

- A variable 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. This means the variable name `ItemsOrdered` is not the same as `itemsordered`.

Table 2-1 The Python key words

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

Displaying Multiple Items with the `print` Statement

If you look back at Program 2-4 you will see that we used the following two `print` statements in lines 3 and 4:

```
print 'I am staying in room number'
print room
```

We used two `print` statements because we needed to display two pieces of data. Line 3 displays the string literal `'I am staying in room number'`, and line 4 displays the value referenced by the `room` variable.

This program can be simplified, however, because Python allows us to display multiple items with one `print` statement. We simply have to separate the items with commas as shown in Program 2-5.

Program 2-5 (*variable_demo3.py*)

```
1 room = 503
2 print 'I am staying in room number', room
```

Program Output

```
I am staying in room number 503
```

The `print` statement in line 2 displays two items: a string literal followed by the value referenced by the `room` variable. Notice that Python automatically printed a space between these two items. When multiple items are printed this way in Python, they will automatically be separated by a space.

Numeric Data Types and Literals

Python uses the `int` data type to store integers, and the `float` data type to store real numbers. Let's look at how Python determines the data type of a number. Many of the programs that you will see will have numeric data written into their code. For example, the following statement, which appears in Program 2-4, has the number 503 written into it.

```
room = 503
```

This statement causes the value 503 to be stored in memory, and it makes the `room` variable reference it. The following statement shows another example. This statement has the number 2.75 written into it.

```
dollars = 2.75
```

This statement causes the value 2.75 to be stored in memory, and it makes the `dollars` variable reference it. A number that is written into a program's code is called a *numeric literal*. When the Python interpreter reads a numeric literal in a program's code, it determines its data type according to the following rules:

- A numeric literal that is written as a whole number with no decimal point is considered an `int`. Examples are 7, 124, and -9.
- A numeric literal that is written with a decimal point is considered a `float`. Examples are 1.5, 3.14159, and 5.0.

So, the following statement causes the number 503 to be stored in memory as an `int`:

```
room = 503
```

And the following statement causes the number 2.75 to be stored in memory as a `float`:

```
dollars = 2.75
```

Storing Strings with the `str` Data Type

In addition to the `int` and `float` data types, Python also has a data type named `str`, which is used for storing strings in memory. The code in Program 2-6 shows how strings can be assigned to variables.

Program 2-6 (*string_variable.py*)

```
1 first_name = 'Kathryn'
2 last_name = 'Marino'
3 print first_name, last_name
```

Program Output

Kathryn Marino

Reading Input from the Keyboard

In this booklet we will use two of Python's built-in functions to read input from the keyboard. A *function* is a piece of prewritten code that performs an operation and then returns a value back to the program. We will use the `input` function to read numeric data from the keyboard, and the `raw_input` function to read strings as input.

Reading Numbers With The `input` Function

Python's `input` function is useful for reading numeric input from the keyboard. You normally use the `input` function in an assignment statement that follows this general format:

```
variable = input(prompt)
```

In the general format, *prompt* is a string that is displayed on the screen. The string's purpose is to instruct the user to enter a value. *variable* is the name of a variable that will reference the data that was entered on the keyboard. Here is an example of a statement that uses the `input` function to read data from the keyboard:

```
hours = input('How many hours did you work? ')
```

When this statement executes, the following things happen:

- The string 'How many hours did you work? ' is displayed on the screen.
- The program pauses and waits for the user to type something on the keyboard, and then press the Enter key.
- When the Enter key is pressed, the data that was typed is assigned to the `hours` variable.

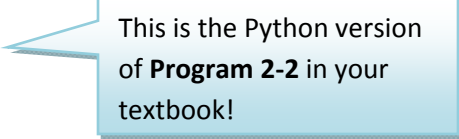
Program 2-7 shows a sample program that uses the input function.

Program 2-7 (input.py)

```
1 age = input('What is your age? ')
2 print 'Here is the value that you entered:'
```

Program Output (with Input Shown in Bold)

```
What is your age? 28 [Enter]
Here is the value that you entered:
28
```



This is the Python version of **Program 2-2** in your textbook!

The statement in line 1 uses the `input` function to read data that is typed on the keyboard. In the sample run, the user typed 28 and then pressed Enter. As a result, the integer value 28 was assigned to the `age` variable.

Take a closer look at the string we used as a prompt, in line 2:

```
'How old are you? '
```

Notice that the last character in the string, inside the quote marks, is a space. We put a space there because the input function does not automatically display a space after the prompt. When the user begins typing characters, they will appear on the screen immediately after the prompt. Making the last character in the prompt a space visually separates the prompt from the user's input on the screen.

When the user enters a number in response to the `input` function, Python determines the number's data type in the same way that it determines a numeric literal's data type: If the number contains no decimal point it is stored in memory as an `int`. If it contains a decimal point it is stored in memory as a `float`.

Reading Strings With The `raw_input` Function

Although the `input` function works well for reading numbers, it is not convenient for reading strings. In order for the `input` function to read data as a string, the user has to enclose the data in quote-marks when he or she types it on the keyboard. Most users are not accustomed to doing this, so it's best to use another function: `raw_input`.

The `raw_input` function works like the `input` function, with one exception: the `raw_input` function retrieves all keyboard input as a string. There is no need for the user to type quote marks around the data that is entered. Program 2-8 shows a sample program that uses the `raw_input` function to read strings.

Program 2-8 (*string_input.py*)

```
1 first_name = raw_input('Enter your first name: ')
2 last_name = raw_input('Enter your last name: ')
3 print 'Hello', first_name, last_name
```

Program Output (with Input Shown in Bold)

```
Enter your first name: Vinny [Enter]
Enter your last name: Brown [Enter]
Hello Vinny Brown
```

Performing Calculations

Table 2-2 lists the math operators that are provided by the Python language.

Table 2-2 Python math operators

Symbol	Operation	Description
+	Addition	Adds two numbers
-	Subtraction	Subtracts one number from another
*	Multiplication	Multiplies one number by another
/	Division	Divides one number by another and gives the quotient
%	Remainder	Divides one number by another and gives the remainder
**	Exponent	Raises a number to a power

Here are some examples of statements that use an arithmetic operator to calculate a value, and assign that value to a variable:

```
total = price + tax
sale = price - discount
population = population * 2
half = number / 2
leftOver = 17 % 3
result = 4**2
```

Program 2-9 shows an example program that performs mathematical calculations (This program is the Python version of pseudocode Program 2-8 in your textbook.)

Program 2-9 (*sale_price.py*)

```
1 original_price = input("Enter the item's original price: ")
2 discount = original_price * 0.2
3 sale_price = original_price - discount
4 print 'The sale price is', sale_price
```

This is the Python version
of **Program 2-8** in your
textbook!

Program Output (With Input Shown in Bold)</TTL>

```
Enter the item's original price: 100.00 [Enter]
The sale price is 80.0
```

In Python, the order of operations and the use of parentheses as grouping symbols works just as described in the textbook.

Integer Division

In Python, when an integer is divided by an integer the result will also be an integer. This behavior is known as *integer division*. For example, look at the following statement:

```
number = 3 / 2
```

Because the numbers 3 and 2 are both treated as integers, Python will throw away (truncate) the fractional part of the result. So, the statement will assign the value 1 to the number variable, not 1.5.

If you want to make sure that a division operation yields a real number, at least one of the operands must be a number with a decimal point or a variable that references a `float` value. For example, we could rewrite the statement as follows:

```
number = 3.0 / 2.0
```

Documenting a Program with Comments

To write a line comment in Python you simply place the `#` symbol where you want the comment to begin. The Python interpreter ignores everything from that point to the end of the line. Here is an example:

```
# This program calculates an employee's gross pay.
```

Chapter 3 Modularizing Programs with Functions

Chapter 3 in your textbook discusses modules as named groups of statements that perform specific tasks in a program. You use modules to break a program down into small, manageable units. In Python, we use *functions* for this purpose. (In Python, the term "module" has a slightly different meaning. A Python module is a file that contains a set of related program elements, such as functions.)

In this chapter we will discuss how to define and call Python functions, use local variables in a function, and pass arguments to a function. We also discuss global variables, and the use of global constants.

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.
```

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.

Let's look at an example of a function. Keep in mind that this is not a complete program. We will show the entire program in a moment.

```
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 `print` statements. Executing the function will cause these `print` 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*. To fully demonstrate how function calling works, look at Program 3-1.

Program 3-1 (*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.
```

When the Python interpreter reads the `def` statement in line 3, a function named `message` is created in memory, containing the block of statements in lines 4 and 5. (A function definition creates a function, but it does not cause the function to execute.) Next, the interpreter encounters the comment in line 7, which is ignored. Then it 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.

Program 3-1 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. Program 3-2 shows an example of a program with two functions: `main` and `show_message`. This is the Python version of Program 3-1 in your textbook, with some extra comments added.

Program 3-2 (*function_demo2.py*)

```
1  # Define the main function.
2  def main():
3      print "I have a message for you."
4      show_message()
5      print "That's all folks!"
6
7  # Define the show_message function.
8  def show_message():
9      print "Hello world"
10
11 # Call the main function.
12 main()
```



This is the Python version
of **Program 3-1** in your
textbook!

Program Output

```
I have a message for you.
Hello world
That's all folks!
```

The `main` function is defined in lines 2 through 5, and the `show_message` function is defined in lines 8 through 9. When the program runs, the statement in line 12 calls the `main` function, which then calls the `show_message` function in line 4.

Indentation in Python

In Python, each line in a block must be indented. As shown in Figure 3-1, the last indented line after a function header is the last line in the function's block.

Figure 3-1 All of the statements in a block are indented

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()
```

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.

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.

Tip: 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.

Note: Blank lines that appear in a block are ignored.

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.) In Chapter 3 of your textbook you learned that a variable's scope is the part of the program in which the variable may be accessed. 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. For example, look at the Program 3-3. In addition to the `main` function, this program has two other functions: `texas` and `california`. These two functions each have a local variable named `birds`.

Program 3-3 (*birds.py*)

```
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.
```

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.

Passing Arguments to Functions

If you want a function to receive arguments when it is called, you must equip the function with one or more parameter variables. A *parameter variable*, often simply called a *parameter*, is a special variable that is assigned the value of an argument when a function is called. Here is an example of a function that has a parameter variable:

```
def double_number(value):  
    result = value * 2  
    print result
```

This function's name is `double_number`. Its purpose 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 `value` 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. Program 3-4 demonstrates the function in a complete program.

Program 3-4 (*pass_arg.py*)

```
1 # Define the main function.  
2 def main():  
3     number = input('Enter a number and I will display that number doubled: ')  
4     double_number(number)  
5  
6 # Define the double_number function.  
7 def double_number(value):  
8     result = value * 2  
9     print result  
10  
11 # Call the main function.  
12 main()
```



This is the Python version of
Program 3-5 in your textbook!

Program Output

```
Enter a number and I will display that number doubled: 20 [Enter]  
40
```

When this program runs, the `main` function is called in line 12. Inside the `main` function, line 3 gets a number from the user and assigns it to the `number` variable. Line 4 calls the `double_number` function passing the `number` variable as an argument.

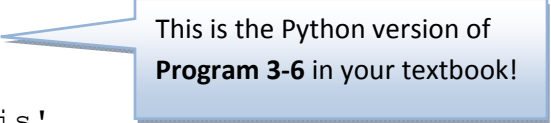
The `double_number` function is defined in lines 26 through 36. The function has a parameter variable named `value`. In line 8 a local variable named `result` is assigned the value of the math expression `value * 2`. In line 35 the value of the `result` variable is displayed.

Passing Multiple Arguments

Often it is useful to pass more than one argument to a function. When you define a function, you must have a parameter variable for each argument that you want passed into the function. Program 3-5 shows an example. This is the Python version of pseudocode Program 3-6 in your textbook.

Program 3-5 (*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()
```



This is the Python version of
Program 3-6 in your textbook!

Program Output

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

Notice that 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 that 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. 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.

Making Changes to Parameters

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. To demonstrate this look at Program 3-6.

Program 3-6 (*change_me.py*)

```
1  # This program demonstrates what happens when you
2  # change the value of a parameter.
3
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
```

The `main` function creates a local variable named `value` in line 5, assigned the value 99. The `print` 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. This means that in the `change_me` function the `arg` parameter will also reference the value 99.

Global Variables

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. For example, look at Program 3-7.

Program 3-7 (*global1.py*)

```
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

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`.

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, as shown in Program 3-14.

Program 3-14 (*global2.py*)

```
1  # Create a global variable.
2  number = 0
3
4  def main():
5      global number
6      number = input('Enter a number: ')
7      show_number()
```

This is the Python version of
Program 3-11 in your textbook!

```
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: 22 [Enter]
The number you entered is 22
```

The assignment statement in line 2 creates a global variable named `number`. Notice that 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

The Python language does not allow you to create true global constants, but 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.