3

STRINGS AND WRITING PROGRAMS

"The only way to learn a new programming language is by writing programs in it."
—Brian Kernighan and Dennis Ritchie,
The C Programming Language

Chapter 2 gave you enough integers and math for now. Python is more than just a calculator. Because cryptography is all about dealing with text values by turning plaintext into ciphertext and back again, you'll learn how to store, combine, and display text on the screen in this chapter. You'll also make your first program, which greets the user with the text "Hello, world!" and lets the user input their name.

TOPICS COVERED IN THIS CHAPTER

- Strings
- String concatenation and replication
- Indexes and slices
- The `print()` function
- Writing source code with IDLE
- Saving and running programs in IDLE
- Comments
- The `input()` function

Working with Text Using String Values

In Python, we work with little chunks of text called string values (or simply *strings*). All of our cipher and hacking programs deal with string values to turn plaintext like `'One if by land, two if by space'` into ciphertext like `'b1rJvsJo!Jyn1q,J702JvsJo!J63nprM'`. The plaintext and ciphertext are represented in our program as string values, and there are many ways in which Python code can manipulate these values.

You can store string values inside variables just as you can with integer and floating-point values. When you type a string, put it between two single quotes (`'`) to show where the string starts and ends. Enter the following into the interactive shell:

```
>>> spam = 'hello'
```

The single quotes are not part of the string value. Python knows that `'hello'` is a string and `spam` is a variable because strings are surrounded by quotes and variable names are not.

If you enter `spam` into the shell, you will see the contents of the `spam` variable (the `'hello'` string):

```
>>> spam = 'hello'
>>> spam
'hello'
```

This is because Python evaluates a variable to the value stored inside it: in this case, the string `'hello'`. Strings can have almost any keyboard character in them. These are all examples of strings:

```
>>> 'hello'
'hello'
```

```
>>> 'KITTENS'
'KITTENS'
>>> ''
''
>>> '7 apples, 14 oranges, 3 lemons'
'7 apples, 14 oranges, 3 lemons'
>>> 'Anything not pertaining to elephants is irrelephant.'
'Anything not pertaining to elephants is irrelephant.'
>>> 'O*&#wY%*&OcfsdYO*&gfC%YO*&%3yc8r2'
'O*&#wY%*&OcfsdYO*&gfC%YO*&%3yc8r2'
```

Notice that the `''` string has zero characters in it; there is nothing between the single quotes. This is known as a *blank string* or *empty string*.

String Concatenation with the + Operator

You can add two string values to create one new string by using the + operator. Doing so is called *string concatenation*. Enter `'Hello,' + 'world!'` into the shell:

```
>>> 'Hello,' + 'world!'
'Hello,world!'
```

Python concatenates *exactly* the strings you tell it to concatenate, so it won't put a space between strings when you concatenate them. If you want a space in the resulting string, there must be a space in one of the two original strings. To put a space between `'Hello,'` and `'world!'`, you can put a space at the end of the `'Hello,'` string and before the second single quote, like this:

```
>>> 'Hello, ' + 'world!'
'Hello, world!'
```

The + operator can concatenate two string values into a new string value (`'Hello, ' + 'world!'` to `'Hello, world!'`), just like it can add two integer values to result in a new integer value (`2 + 2` to `4`). Python knows what the + operator should do because of the data types of the values. As you learned in Chapter 2, the data type of a value tells us (and the computer) what kind of data the value is.

You can use the + operator in an expression with two or more strings or integers as long as the data types match. If you try to use the operator with one string and one integer, you'll get an error. Enter this code into the interactive shell:

```
>>> 'Hello' + 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
>>> 'Hello' + '42'
'Hello42'
```

The first line of code causes an error because 'Hello' is a string and 42 is an integer. But in the second line of code, '42' is a string, so Python concatenates it.

String Replication with the * Operator

You can also use the * operator on a string and an integer to do *string replication*. This replicates (that is, repeats) a string by however many times the integer value is. Enter the following into the interactive shell:

```
❶ >>> 'Hello' * 3
'HelloHelloHello'
>>> spam = 'Abcdef'
❷ >>> spam = spam * 3
>>> spam
'AbcdefAbcdefAbcdef'
```

To replicate a string, type the string, then the * operator, and then the number of times you want the string to repeat ❶. You can also store a string, like we've done with the spam variable, and then replicate the variable instead ❷. You can even store a replicated string back into the same variable or a new variable.

As you saw in Chapter 2, the * operator can work with two integer values to multiply them. But it can't work with two string values, which would cause an error, like this:

```
>>> 'Hello' * 'world!'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

String concatenation and string replication show that operators in Python can do different tasks based on the data types of the values they operate on. The + operator can do addition or string concatenation. The * operator can do multiplication or string replication.

Getting Characters from Strings Using Indexes

Your encryption programs often need to get a single character from a string, which you can accomplish through indexing. With *indexing*, you add square brackets [and] to the end of a string value (or a variable containing a string) with a number between them to access one character. This number is called the *index*, and it tells Python which position in the string has the character you want. Python indexes start at 0, so the index of the first character in a string is 0. The index 1 is for the second character, the index 2 is for the third character, and so on.

Enter the following into the interactive shell:

```
>>> spam = 'Hello'
>>> spam[0]
'H'
```

```
>>> spam[1]
'e'
>>> spam[2]
'l'
```

Notice that the expression `spam[0]` evaluates to the string value `'H'`, because `H` is the first character in the string `'Hello'` and indexes start at `0`, not `1` (see Figure 3-1).

You can use indexing with a variable containing a string value, as we did with the previous example, or a string value by itself, like this:

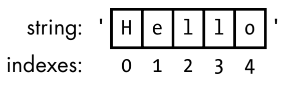


Figure 3-1: The string `'Hello'` and its indexes

```
>>> 'Zophie'[2]
'p'
```

The expression `'Zophie'[2]` evaluates to the third string value, which is a `'p'`. This `'p'` string is just like any other string value and can be stored in a variable. Enter the following into the interactive shell:

```
>>> eggs = 'Zophie'[2]
>>> eggs
'p'
```

If you enter an index that is too large for the string, Python displays an `"index out of range"` error message, as you can see in the following code:

```
>>> 'Hello'[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

There are five characters in the string `'Hello'`, so if you try to use the index `10`, Python displays an error.

Negative Indexes

Negative indexes start at the end of a string and go backward. The negative index `-1` is the index of the *last* character in a string. The index `-2` is the index of the second to last character, and so on, as shown in Figure 3-2.

Enter the following into the interactive shell:

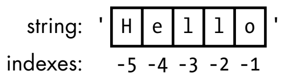


Figure 3-2: The string `'Hello'` and its negative indexes

```
>>> 'Hello'[-1]
'o'
>>> 'Hello'[-2]
'l'
>>> 'Hello'[-3]
'l'
```

```
>>> 'Hello'[-4]
'e'
>>> 'Hello'[-5]
'H'
>>> 'Hello'[0]
'H'
```

Notice that -5 and 0 are the indexes for the same character. Most of the time, your code will use positive indexes, but sometimes it's easier to use negative ones.

Getting Multiple Characters from Strings Using Slices

If you want to get more than one character from a string, you can use slicing instead of indexing. A *slice* also uses the [and] square brackets but has two integer indexes instead of one. The two indexes are separated by a colon (:) and tell Python the index of the first and last characters in the slice. Enter the following into the interactive shell:

```
>>> 'Howdy'[0:3]
'How'
```

The string that the slice evaluates to begins at the first index value and goes up to, but does not include, the second index value. Index 0 of the string value 'Howdy' is H and index 3 is d. Because a slice goes up to but does not include the second index, the slice 'Howdy'[0:3] evaluates to the string value 'How'.

Enter the following into the interactive shell:

```
>>> 'Hello, world!'[0:5]
'Hello'
>>> 'Hello, world!'[7:13]
'world!'
>>> 'Hello, world!'[-6:-1]
'world'
>>> 'Hello, world!'[7:13][2]
'r'
```

Notice that the expression 'Hello, world!'[7:13][2] first evaluates the list slice to 'world!'[2] and then further evaluates to 'r'.

Unlike indexes, slicing never gives you an error if you give too large an index for the string. It'll just return the widest matching slice it can:

```
>>> 'Hello'[0:999]
'Hello'
>>> 'Hello'[2:999]
'llo'
>>> 'Hello'[1000:2000]
''
```

The expression 'Hello'[1000:2000] returns a blank string because the index 1000 is after the end of the string, so there are no possible characters this slice could include. Although our examples don't show this, you can also slice strings stored in variables.

Blank Slice Indexes

If you omit the first index of a slice, Python will automatically use index 0 for the first index. The expressions 'Howdy'[0:3] and 'Howdy'[:3] evaluate to the same string:

```
>>> 'Howdy'[:3]
'How'
>>> 'Howdy'[0:3]
'How'
```

If you omit the second index, Python will automatically use the rest of the string starting from the first index:

```
>>> 'Howdy'[2:]
'wdy'
```

You can use blank indexes in many different ways. Enter the following into the shell:

```
>>> myName = 'Zophie the Fat Cat'
>>> myName[-7:]
'Fat Cat'
>>> myName[:10]
'Zophie the'
>>> myName[7:]
'the Fat Cat'
```

As you can see, you can even use negative indexes with a blank index. Because -7 is the starting index in the first example, Python counts backward seven characters from the end and uses that as its starting index. Then it returns everything from that index to the end of the string because of the second blank index.

Printing Values with the print() Function

Let's try another type of Python instruction: a print() function call. Enter the following into the interactive shell:

```
>>> print('Hello!')
Hello!
>>> print(42)
42
```

A *function* (like print() in this example) has code inside it that performs a task, such as printing values onscreen. Many different functions come with Python and can perform useful tasks for you. To *call* a function means to execute the code inside the function.

The instructions in this example pass a value to print() between the parentheses, and the print() function prints the value to the screen. The values that are passed when a function is called are *arguments*. When you write programs, you'll use print() to make text appear on the screen.

You can pass an expression to print() instead of a single value. This is because the value that is actually passed to print() is the evaluated value of that expression. Enter this string concatenation expression into the interactive shell:

```
>>> spam = 'Al'
>>> print('Hello, ' + spam)
Hello, Al
```

The 'Hello, ' + spam expression evaluates to 'Hello, ' + 'Al', which then evaluates to the string value 'Hello, Al'. This string value is what is passed to the print() call.

Printing Escape Characters

You might want to use a character in a string value that would confuse Python. For example, you might want to use a single quote character as part of a string. But you'd get an error message because Python thinks that single quote is the quote ending the string value and the text after it is bad Python code, instead of the rest of the string. Enter the following into the interactive shell to see the error in action:

```
>>> print('Al's cat is named Zophie.')
SyntaxError: invalid syntax
```

To use a single quote in a string, you need to use an *escape character*. An escape character is a backslash character followed by another character—for example, \t, \n, or \'. The slash tells Python that the character after the slash has a special meaning. Enter the following into the interactive shell.

```
>>> print('Al\'s cat is named Zophie.')
Al's cat is named Zophie.
```

Now Python will know the apostrophe is a character in the string value, not Python code marking the end of the string.

Table 3-1 shows some escape characters you can use in Python.

Table 3-1: Escape Characters

Escape character	Printed result
`\\`	Backslash (\)
`\'`	Single quote (')
`\"`	Double quote (")
`\n`	Newline
`\t`	Tab

The backslash always precedes an escape character. Even if you just want a backslash in your string, you can't add a backslash alone because Python will interpret the next character as an escape character. For example, this line of code wouldn't work correctly:

```
>>> print('It is a green\teal color.')
It is a green    eal color.
```

The `'t'` in `'teal'` is identified as an escape character because it comes after a backslash. The escape character `\t` simulates pushing the TAB key on your keyboard.

Instead, enter this code:

```
>>> print('It is a green\\teal color.')
It is a green\teal color.
```

This time the string will print as you intended, because putting a second backslash in the string makes the backslash the escape character.

Quotes and Double Quotes

Strings don't always have to be between two single quotes in Python. You can use double quotes instead. These two lines print the same thing:

```
>>> print('Hello, world!')
Hello, world!
>>> print("Hello, world!")
Hello, world!
```

But you can't mix single and double quotes. This line gives you an error:

```
>>> print('Hello, world!")
SyntaxError: EOL while scanning string literal
```

I prefer to use single quotes because they're a bit easier to type than double quotes and Python doesn't care either way.

But just like you have to use the escape character \' to have a single quote in a string surrounded by single quotes, you need the escape character \" to have a double quote in a string surrounded by double quotes. For example, look at these two lines:

```
>>> print('Al\'s cat is Zophie. She says, "Meow."')
Al's cat is Zophie. She says, "Meow."
>>> print("Zophie said, \"I can say things other than 'Meow' you know.\"")
Zophie said, "I can say things other than 'Meow' you know."
```

You don't need to escape double quotes in single-quote strings, and you don't need to escape single quotes in double-quote strings. The Python interpreter is smart enough to know that if a string starts with one kind of quote, the other kind of quote doesn't mean the string is ending.

Writing Programs in IDLE's File Editor

Until now, you've been entering instructions one at a time into the interactive shell. But when you write programs, you'll enter several instructions and have them run without waiting on you for the next one. It's time to write your first program!

The name of the software program that provides the interactive shell is called IDLE (**I**ntegrated **D**eve**L**opment **E**nvironment). In addition to the interactive shell, IDLE also has a *file editor*, which we'll open now.

At the top of the Python shell window, select **File ▸ New Window**. A new blank window, the file editor, will appear for you to enter a program, as shown in Figure 3-3. The bottom-right corner of the file editor window shows you what line and column the cursor currently is on.

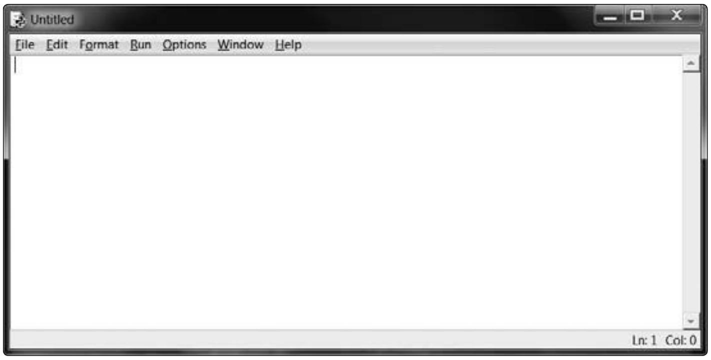


Figure 3-3: The file editor window with the cursor at line 1, column 0

You can tell the difference between the file editor window and the interactive shell window by looking for the >>> prompt. The interactive shell always displays the prompt, and the file editor doesn't.

Source Code for the "Hello, World!" Program

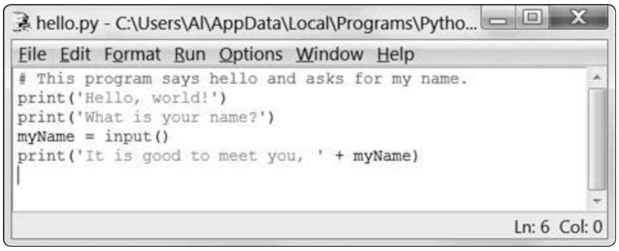
Traditionally, programmers who are learning a new language make their first program display the text `"Hello, world!"` on the screen. We'll create our own "Hello, world!" program next by entering text into the new file editor window. We call this text the program's *source code* because it contains the instructions that Python will follow to determine exactly how the program should behave.

You can download the "Hello, world!" source code from *https://www.nostarch.com/crackingcodes/*. If you get errors after entering this code, compare it to the book's code using the online diff tool (see "Checking Your Source Code with the Online Diff Tool" next). Remember that you don't type the line numbers; they only appear in this book to aid explanation.

hello.py

```
1. # This program says hello and asks for my name.
2. print('Hello, world!')
3. print('What is your name?')
4. myName = input()
5. print('It is good to meet you, ' + myName)
```

The IDLE program will display different types of instructions in different colors. When you're done entering this code, the window should look like Figure 3-4.


hello.py - C:\Users\Al\AppData\Local\Programs\Pytho...
File Edit Format Run Options Window Help
Ln: 6 Col: 0

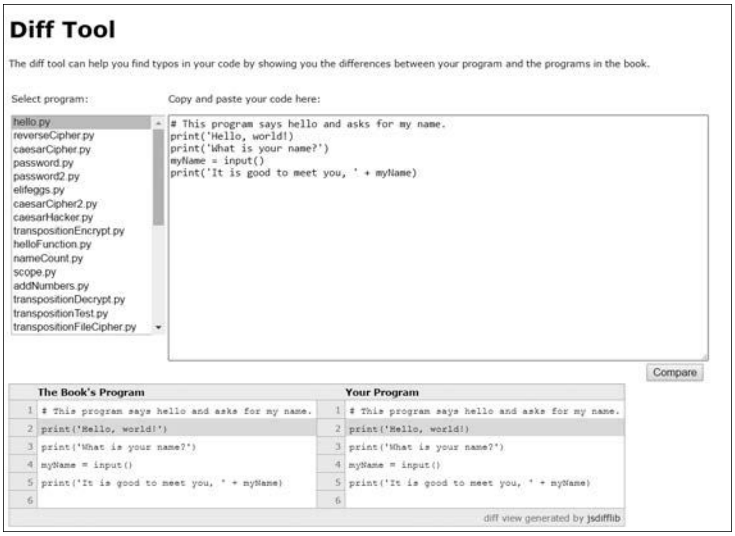
```
# This program says hello and asks for my name.
print('Hello, world!')
print('What is your name?')
myName = input()
print('It is good to meet you, ' + myName)
```

Figure 3-4: The file editor window will look like this after you enter the code.

Checking Your Source Code with the Online Diff Tool

Even though you could copy and paste or download the *hello.py* code from this book's website, you should still type this program manually. Doing so will give you more familiarity with the code in the program. However, you might make some mistakes while typing it into the file editor.

To compare the code you typed to the code in this book, use the online diff tool shown in Figure 3-5. Copy the text of your code and then navigate to the diff tool on the book's website at *https://www.nostarch.com/crackingcodes/*. Select the *hello.py* program from the drop-down menu. Paste your code into the text field on this web page and click the **Compare** button. The diff tool shows any differences between your code and the code in this book. This is an easy way to find any typos causing errors in your program.



The diff tool can help you find typos in your code by showing you the differences between your program and the programs in the book.

Select program:

hello.py
reverseCipher.py
caesarCipher.py
password.py
password2.py
elifeggs.py
caesarCipher2.py
caesarHacker.py
transpositionEncrypt.py
helloFunction.py
nameCount.py
scope.py
addNumbers.py
transpositionDecrypt.py
transpositionTest.py
transpositionFileCipher.py

Copy and paste your code here:

```
# This program says hello and asks for my name.
print('Hello, world!)
print('What is your name?')
myName = input()
print('It is good to meet you, ' + myName)
```

The Book's Program	Your Program
`# This program says hello and asks for my name.`	`# This program says hello and asks for my name.`
`print('Hello, world!')`	`print('Hello, world!)`
`print('What is your name?')`	`print('What is your name?')`
`myName = input()`	`myName = input()`
`print('It is good to meet you, ' + myName)`	`print('It is good to meet you, ' + myName)`

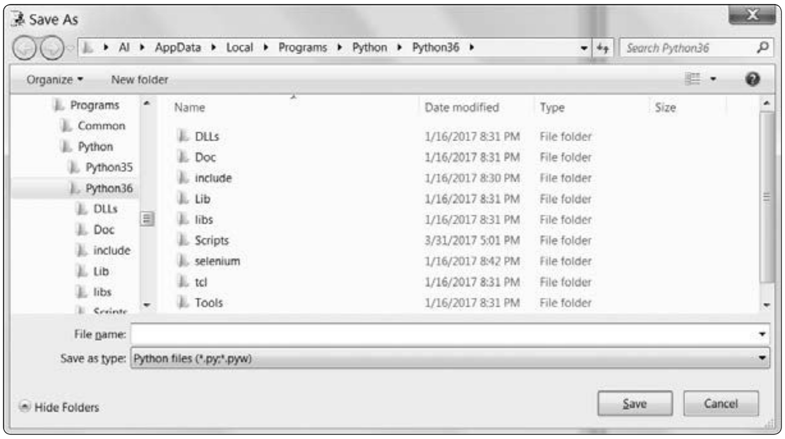
Figure 3-5: The online diff tool

Using IDLE to Access Your Program Later

When you write programs, you might want to save them and come back to them later, especially after you've typed a very long program. IDLE has features for saving and opening programs just like a word processer has features to save and reopen your documents.

Saving Your Program

After you've entered your source code, save it so you won't have to retype it each time you want to run it. Choose **File ▸ Save As** from the menu at the top of the file editor window. The Save As dialog should open, as shown in Figure 3-6. Enter **`hello.py`** in the **File Name** field and click **Save**.



Name	Date modified	Type	Size
DLLs	1/16/2017 8:31 PM	File folder	
Doc	1/16/2017 8:31 PM	File folder	
include	1/16/2017 8:30 PM	File folder	
Lib	1/16/2017 8:31 PM	File folder	
libs	1/16/2017 8:31 PM	File folder	
Scripts	3/31/2017 5:01 PM	File folder	
selenium	1/16/2017 8:42 PM	File folder	
tcl	1/16/2017 8:31 PM	File folder	
Tools	1/16/2017 8:31 PM	File folder	

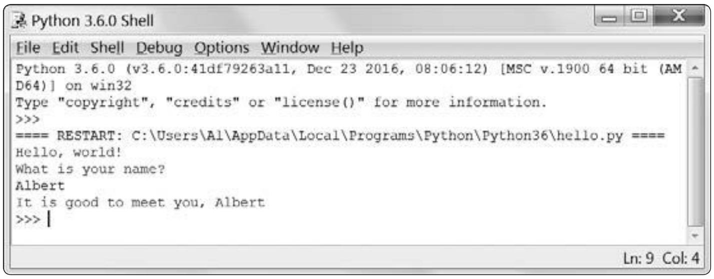
Figure 3-6: Saving the program

You should save your programs often as you type them so you won't lose your work if the computer crashes or if you accidentally exit from IDLE. As a shortcut, you can press CTRL-S on Windows and Linux or ⌘-S on macOS to save your file.

Running Your Program

Now it's time to run your program. Select **Run ▸ Run Module** or just press the F5 key on your keyboard. Your program should run in the shell window that appeared when you first started IDLE. Remember that you must press F5 from the file editor's window, not the interactive shell's window.

When the program asks for your name, enter it, as shown in Figure 3-7.



```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
==== RESTART: C:\Users\Al\AppData\Local\Programs\Python\Python36\hello.py ====
Hello, world!
What is your name?
Albert
It is good to meet you, Albert
>>> |
```

Figure 3-7: The interactive shell looks like this when running the "Hello, world!" program.

Now when you press ENTER, the program should greet you (the *user*, that is, the one using the program) by name. Congratulations! You've written your first program. You are now a beginning computer programmer. (If you like, you can run this program again by pressing F5 again.)

If instead you get an error that looks like this, it means you are running the program with Python 2 instead of Python 3:

```
Hello, world!
What is your name?
Albert
Traceback (most recent call last):
  File "C:/Python27/hello.py", line 4, in <module>
    myName = input()
  File "<string>", line 1, in <module>
NameError: name 'Albert' is not defined
```

The error is caused by the `input()` function call, which behaves differently in Python 2 and 3. Before continuing, install Python 3 by following the instructions in "Downloading and Installing Python" on page xxv.

Opening the Programs You've Saved

Close the file editor by clicking the X in the top corner. To reload a saved program, choose **File ▸ Open** from the menu. Do that now, and in the window that appears, choose *hello.py*. Then click the **Open** button. Your saved *hello.py* program should open in the file editor window.

How the "Hello, World!" Program Works

Each line in the "Hello, world!" program is an instruction that tells Python exactly what to do. A computer program is a lot like a recipe. Do the first step first, then the second, and so on until you reach the end. When the program follows instructions step-by-step, we call it the *program execution*, or just the *execution*.

Each instruction is followed in sequence, beginning from the top of the program and working down the list of instructions. The execution starts at the first line of code and then moves downward. But the execution can also skip around instead of just going from top to bottom; you'll find out how to do this in Chapter 4.

Let's look at the "Hello, world!" program one line at a time to see what it's doing, beginning with line 1.

Comments

Any text following a *hash mark* (`#`) is a comment:

```
1. # This program says hello and asks for my name.
```

Comments are not for the computer but instead are for you, the programmer. The computer ignores them. They're used to remind you what the program does or to tell others who might look at your code what your code does.

Programmers usually put a comment at the top of their code to give the program a title. The IDLE program displays comments in red text to help them stand out. Sometimes, programmers will put a `#` in front of a line of code to temporarily skip it while testing a program. This is called *commenting out* code, and it can be useful when you're trying to figure out why a program doesn't work. You can remove the `#` later when you're ready to put the line back in.

Printing Directions to the User

The next two lines display directions to the user with the `print()` function. A function is like a mini-program inside your program. The great benefit of using functions is that we only need to know what the function does, not how it does it. For instance, you need to know that `print()` displays text onscreen, but you don't need to know the exact code inside the function that does this.

A function call is a piece of code that tells the program to run the code inside a function.

Line 2 of *hello.py* is a call to print() (with the string to be printed inside the parentheses). Line 3 is another print() call. This time the program displays 'What is your name?'

```
2. print('Hello, world!')
3. print('What is your name?')
```

We add parentheses to the end of function names to make it clear that we're referring to a function named print(), not a variable named print. The parentheses at the end of the function tell Python we're using a function, much as the quotes around the number '42' tell Python that we're using the string '42', not the integer 42.

Taking a User's Input

Line 4 has an assignment statement with a variable (myName) and the new function call input():

```
4. myName = input()
```

When input() is called, the program waits for the user to type in some text and press ENTER. The text string that the user enters (their name) becomes the string value that is stored in myName.

Like expressions, function calls evaluate to a single value. The value that the call evaluates to is called the *return value.* (In fact, we can also use the word "returns" to mean the same thing as "evaluates" for function calls.) In this case, the return value of input() is the string that the user entered, which should be their name. If the user entered Albert, the input() call evaluates to (that is, returns) the string 'Albert'.

Unlike print(), the input() function doesn't need any arguments, which is why there is nothing between the parentheses.

The last line of the code in *hello.py* is another print() call:

```
5. print('It is good to meet you, ' + myName)
```

For line 5's print() call, we use the plus operator (+) to concatenate the string 'It is good to meet you, ' and the string stored in the myName variable, which is the name that the user input into the program. This is how we get the program to greet the user by name.

Ending the Program

When the program executes the last line, it stops. At this point it has *terminated* or *exited,* and all the variables are forgotten by the computer, including the string stored in myName. If you try running the program again and entering a different name, it will print that name.

```
Hello, world!
What is your name?
Zophie
It is good to meet you, Zophie
```

Remember that the computer only does exactly what you program it to do. In this program, it asks you for your name, lets you enter a string, and then says hello and displays the string you entered.

But computers are dumb. The program doesn't care if you enter your name, someone else's name, or just something silly. You can type in anything you want, and the computer will treat it the same way:

```
Hello, world!
What is your name?
poop
It is good to meet you, poop
```

Summary

Writing programs is just about knowing how to speak the computer's language. You learned a bit about how to do this in Chapter 2, and now you've put together several Python instructions to make a complete program that asks for the user's name and greets that user.

In this chapter, you learned several new techniques to manipulate strings, like using the + operator to concatenate strings. You can also use indexing and slicing to create a new string from part of a different string.

The rest of the programs in this book will be more complex and sophisticated, but they'll all be explained line by line. You can always enter instructions into the interactive shell to see what they do before you put them into a complete program.

Next, we'll start writing our first encryption program: the reverse cipher.

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. If you assign `spam = 'Cats'`, what do the following lines print?

```
spam + spam + spam
spam * 3
```

2. What do the following lines print?

```
print("Dear Alice,\nHow are you?\nSincerely,\nBob")
print('Hello' + 'Hello')
```

3. If you assign `spam = 'Four score and seven years is eighty seven years.'`, what would each of the following lines print?

```
print(spam[5])
print(spam[-3])
print(spam[0:4] + spam[5])
print(spam[-3:-1])
print(spam[:10])
print(spam[-5:])
print(spam[:])
```

4. Which window displays the >>> prompt, the interactive shell or the file editor?
5. What does the following line print?

```
#print('Hello, world!')
```