INVENT YOUR OWN COMPUTER GAMES WITH PYTHON

AL SWEIGART

INVENT YOUR OWN COMPUTER GAMES WITH PYTHON

AL SWEIGART

Early Access edition, 9/9/16

ISBN-10: 1-59327-795-4
ISBN-13: 978-1-59327-795-6

Publisher: William Pollock
Production Editor: Laurel Chun
Cover Illustration: Josh Ellingson
Developmental Editor: Jan Cash
Technical Reviewer: Ari Lacenski
Copyeditor: Rachel Monaghan

BRIEF CONTENTS

1 THE INTERACTIVE SHELL

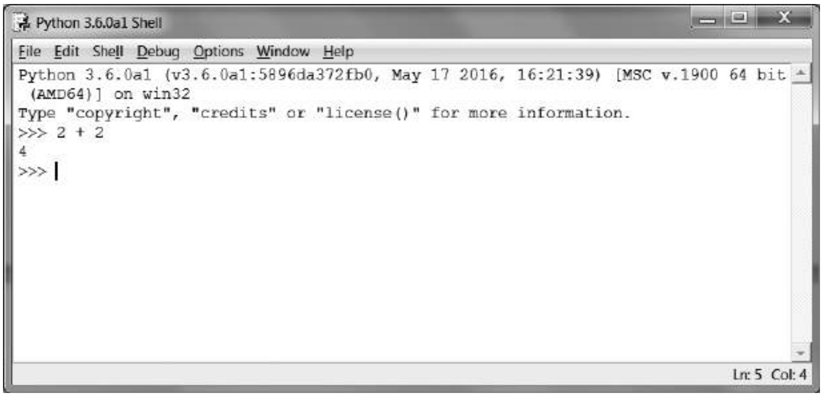
Before you can make games, you need to learn a few basic programming concepts. You'll start in this chapter by learning how to use Python's interactive shell.

TOPICS COVERED IN THIS CHAPTER

- Operators
- Integers and floating-point numbers
- Expressions
- Values
- Evaluating expressions
- Storing values in variables

Some Simple Math

Open IDLE by following the steps in the Introduction. First you'll use Python to solve some simple math problems. The interactive shell can work just like a calculator. Type **`2 + 2`** into the interactive shell at the >>> prompt and press ENTER. (On some keyboards, this key is RETURN.) Figure 1-1 shows how this math problem looks in the interactive shell—notice how it responds with the number `4`.



```
Python 3.6.0a1 (v3.6.0a1:5896da372fb0, May 17 2016, 16:21:39) [MSC v.1900 64 bit
 (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 2 + 2
4
>>> 
```

Figure 1-1: Entering `2 + 2` into the interactive shell

This math problem is a simple programming instruction. The plus sign (`+`) tells the computer to add the numbers `2` and `2`. The computer does this on the next line and responds with the number `4`. Table 1-1 lists the other math symbols available in Python.

Table 1-1: Math Operators

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division

The minus sign (`-`) subtracts numbers, the asterisk (`*`) multiplies numbers, and the slash (`/`) divides numbers. When used in this way, `+`, `-`, `*`, and `/` are called *operators*. Operators tell Python what to do with the numbers surrounding them.

Integers and Floating-Point Numbers

Integers (or *ints* for short) are whole numbers such as `4`, `99`, and `0`. *Floating-point numbers* (or *floats* for short) are fractions or numbers with decimal points like `3.5`, `42.1`, and `5.0`. In Python, `5` is an integer, but `5.0` is a float.

These numbers are called *values*. (Later we will learn about other kinds of values besides numbers.) In the math problem you just typed, `2` and `2` are integer values.

Expressions

The math problem `2 + 2` is an example of an *expression*. As Figure 1-2 shows, expressions are made up of values (the numbers) connected by operators (the math signs) that produce a new value the code can use. Computers can solve millions of expressions in seconds.

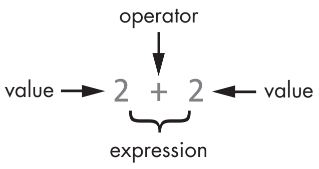


Figure 1-2: An expression is made up of values and operators.

Try entering some of these expressions into the interactive shell, pressing ENTER after each one:

```
>>> 2+2+2+2+2
10
>>> 8*6
48
>>> 10-5+6
11
>>> 2   +          2
4
```

These expressions all look like regular math equations, but notice all the spaces in the `2   +          2` example. In Python, you can add any number of spaces between the values and operators. However, you must always start instructions at the beginning of the line (with no spaces) when entering them into the interactive shell.

Evaluating Expressions

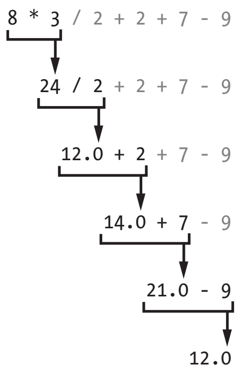
When a computer solves the expression `10 + 5` and returns the value `15`, it has *evaluated* the expression. Evaluating an expression *reduces the expression to a single value*, just like solving a math problem reduces the problem to a single number: the answer. For example, the expressions `10 + 5` and `10 + 3 + 2` both evaluate to `15`.

When Python evaluates an expression, it follows an order of operations just like in mathematics. There are just a few rules:

- Parts of the expression inside parentheses are evaluated first.
- Multiplication and division are done before addition and subtraction.
- The evaluation is performed left to right.

The expression `1 + 2 * 3 + 4` evaluates to `11`, not `13`, because `2 * 3` is evaluated first. If the expression were `(1 + 2) * (3 + 4)` it would evaluate to `21`, because the `(1 + 2)` and `(3 + 4)` inside parentheses are evaluated before multiplication.

Expressions can be of any size, but they will always evaluate to a single value. Even single values are expressions. For example, the expression `15` evaluates to the value `15`. The expression `8 * 3 / 2 + 2 + 7 - 9` will evaluate down to the value `12.0` through the following steps:



Even though the computer is performing all of these steps, you don't see that in the interactive shell. The interactive shell just shows you the result:

```
>>> 8 * 3 / 2 + 2 + 7 - 9
12.0
```

Notice that the `/` division operator evaluates to a float value, as in `24 / 2` evaluating to `12.0`. Math operations with even one float value also evaluate to float values, so `12.0 + 2` evaluates to `14.0`.

Syntax Errors

If you enter `5 +` into the interactive shell, you'll get the following error message:

```
>>> 5 +
SyntaxError: invalid syntax
```

This error happened because `5 +` isn't an expression. Expressions have values connected by operators, and the `+` operator expects a value before *and* after it. An error message appears when an expected value is missing.

`SyntaxError` means Python doesn't understand the instruction because you typed it incorrectly. Computer programming isn't just about giving the computer instructions to follow, but also knowing how to give it those instructions correctly.

Don't worry about making mistakes, though. Errors won't damage your computer. Just retype the instruction correctly into the interactive shell at the next >>> prompt.

Storing Values in Variables

When an expression evaluates to a value, you can use that value later by storing it in a *variable*. Think of variables as a box that can hold a value.

An *assignment statement* will store a value inside a variable. Type a name for the variable, followed by the equal sign (=), which is called the *assignment operator*, and then the value to store in the variable. For example, enter **spam = 15** into the interactive shell:

```
>>> spam = 15
>>>
```

The spam variable's box now stores the value 15, as shown in Figure 1-3.

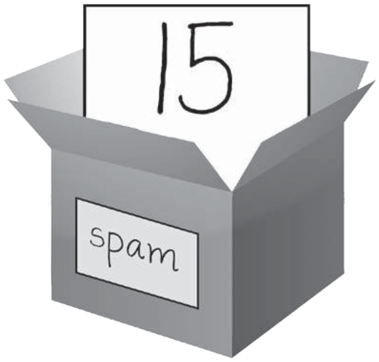


Figure 1-3: Variables are like boxes that can hold values.

When you press ENTER you won't see anything in response. In Python, you know the instruction was successful if no error message appears. The >>> prompt will appear so you can enter the next instruction.

Unlike expressions, *statements* are instructions that do not evaluate to any value. This is why there's no value displayed on the next line in the interactive shell after spam = 15. If you're confused about which instructions are expressions and which are statements, remember that expressions evaluate to a single value. Any other kind of instruction is a statement.

Variables store values, not expressions. For example, consider the expressions in the statements spam = 10 + 5 and spam = 10 + 7 - 2. They both evaluate to 15. The end result is the same: both assignment statements store the value 15 in the variable spam.

A good variable name describes the data it contains. Imagine that you moved to a new house and labeled all of your moving boxes as "Stuff." You'd never find anything! The variable names `spam`, `eggs`, and `bacon` are generic names used for the examples in this book.

The first time a variable is used in an assignment statement, Python will create that variable. To check what value is in a variable, enter the variable name into the interactive shell:

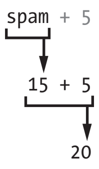
```
>>> spam = 15
>>> spam
15
```

The expression `spam` evaluates to the value inside the `spam` variable: `15`.

You can also use variables in expressions. Try entering the following in the interactive shell:

```
>>> spam = 15
>>> spam + 5
20
```

You set the value of the variable `spam` to `15`, so typing `spam + 5` is like typing the expression `15 + 5`. Here are the steps of `spam + 5` being evaluated:



You cannot use a variable before an assignment statement creates it. If you try to, Python will give you a `NameError` because no such variable by that name exists yet. Mistyping the variable name also causes this error:

```
>>> spam = 15
>>> spma
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    spma
NameError: name 'spma' is not defined
```

The error appeared because there's a `spam` variable but no `spma` variable.

You can change the value stored in a variable by entering another assignment statement. For example, enter the following into the interactive shell:

```
>>> spam = 15
>>> spam + 5
20
>>> spam = 3
>>> spam + 5
8
```

When you first enter spam + 5, the expression evaluates to 20 because you stored 15 inside spam. However, when you enter spam = 3, the value 15 in the variable's box is replaced, or *overwritten*, with the value 3 since the variable can hold only one value at a time. Now when you enter spam + 5, the expression evaluates to 8 because the value of spam is now 3. Overwriting is like taking a value out of the variable's box to put a new value in, as shown in Figure 1-4.

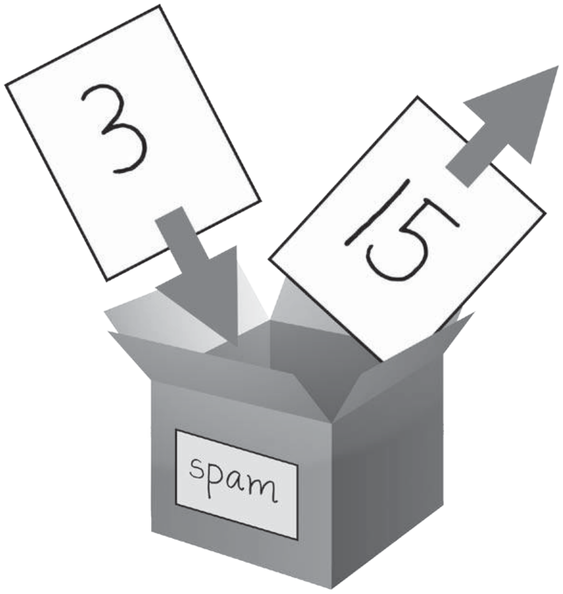


Figure 1-4: The 15 value in spam being overwritten by the 3 value.

You can even use the value in the spam variable to assign a new value to spam:

```
>>> spam = 15
>>> spam = spam + 5
20
```

The assignment statement spam = spam + 5 is like saying, "the new value of the spam variable will be the current value of spam plus five." To keep increasing the value in spam by 5 several times, enter the following into the interactive shell:

```
>>> spam = 15
>>> spam = spam + 5
>>> spam = spam + 5
>>> spam = spam + 5
>>> spam
30
```

In this example, you assign spam a value of 15 in the first statement. In the next statement, you add 5 to the value of spam and assign spam the new value spam + 5, which evaluates to 20. When you do this three times, spam evaluates to 30.

So far we've only looked at one variable, but you can create as many variables as you need in your programs. For example, let's assign different values to two variables named eggs and bacon, like so:

```
>>> bacon = 10
>>> eggs = 15
```

Now the bacon variable has 10 inside it, and the eggs variable has 15 inside it. Each variable is its own box with its own value, as shown in Figure 1-5.

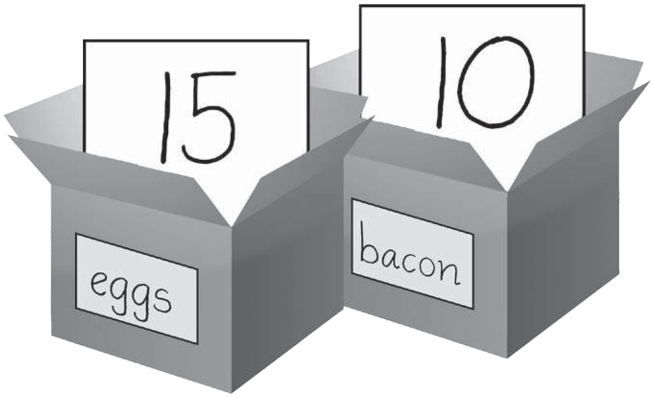


Figure 1-5: The bacon and eggs variables each store values.

Enter spam = bacon + eggs into the interactive shell, then check the new value of spam:

```
>>> bacon = 10
>>> eggs = 15
>>> spam = bacon + eggs
>>> spam
25
```

The value in spam is now 25. When you add bacon and eggs you are adding their values, which are 10 and 15, respectively. Variables contain values, not expressions, so the spam variable was assigned the value 25, not the expression bacon + eggs. After the spam = bacon + eggs assignment statement, changing bacon or eggs does not affect spam.

Summary

In this chapter, you learned the basics of writing Python instructions. Because computers don't have common sense and only understand specific instructions, Python needs you to tell it exactly what to do.

Expressions are values (such as `2` or `5`) combined with operators (such as `+` or `-`). Python can evaluate expressions—that is, reduce the expression to a single value. You can store values inside of variables so that your program can remember those values and use them later.

There are a few other types of operators and values in Python. In the next chapter, you'll go over some more basic concepts and write your first program. You'll learn about working with text in expressions. Python isn't limited to just numbers; it's more than a calculator!

2
WRITING PROGRAMS

That's enough math for now. Now let's see what Python can do with text. Almost all programs display text to the user, and the user enters text into your programs through the keyboard. In this chapter you'll make your first program, which does both of these things. You'll learn how to store text in variables, combine text, and display text on the screen. The program you'll create displays the greeting `Hello world!` and asks for the user's name.

TOPICS COVERED IN THIS CHAPTER

- Strings
- String concatenation
- Data types (such as strings or integers)
- Using the file editor to write programs
- Saving and running programs in IDLE
- Flow of execution
- Comments
- The `print()` function
- The `input()` function
- Case sensitivity

Strings

In Python, text values are called *strings*. String values can be used just like integer or float values. You can store strings in variables. In code, string values start and end with a single quote, `'`. Enter this code into the interactive shell:

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

The single quotes tell Python where the string begins and ends. They are not part of the string value's text. Now if you enter `spam` into the interactive shell, you'll see the contents of the `spam` variable. Remember, Python evaluates variables as the value stored inside the variable. In this case, this is the string `'hello'`:

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

Strings can have any keyboard character in them and can be as long as you want. These are all examples of strings:

```
'hello'
'Hi there!'
'KITTENS'
'7 apples, 14 oranges, 3 lemons'
'Anything not pertaining to elephants is irrelephant.'
'A long time ago, in a galaxy far, far away...'
'O*&#wY%*&OCfsdYO*&gfC%YO*&%3yc8r2'
```

String Concatenation

You can combine string values with operators to make expressions, just like you did with integer and float values. When you combine two strings with the + operator, it's called *string concatenation*. Enter `'Hello' + 'World!'` into the interactive shell:

```
>>> 'Hello' + 'World!'
'HelloWorld!'
```

The expression evaluates to a single string value, `'HelloWorld!'`. There is no space between the words because there was no space in either of the two concatenated strings, unlike in this example:

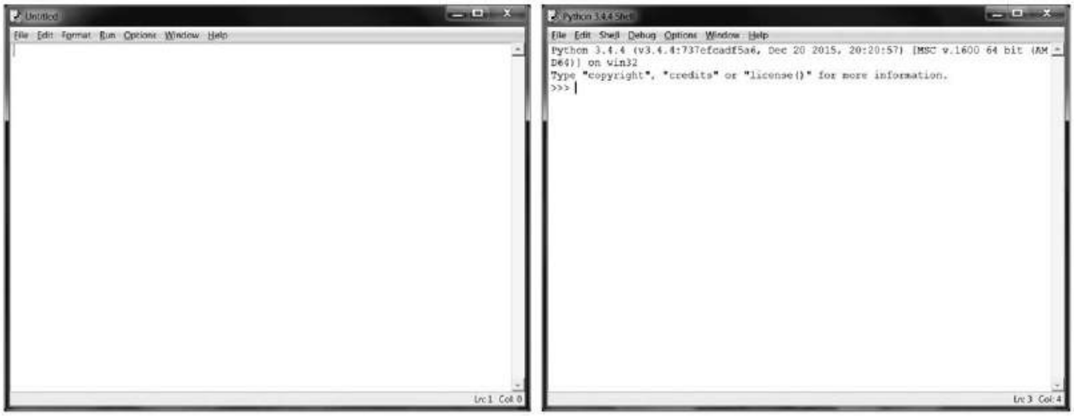
```
>>> 'Hello ' + 'World!'
'Hello World!'
```

The + operator works differently on string and integer values because they are different *data types*. All values have a data type. The data type of the value `'Hello'` is a string. The data type of the value `5` is an integer. The data type tells Python what operators should do when evaluating expressions. The + operator concatenates string values, but adds integer and float values.

Writing Programs in IDLE's File Editor

Until now, you've been typing instructions into IDLE's interactive shell one at a time. When you write programs, though, you enter several instructions and have them run all at once, which is what you'll do next. It's time to write your first program!

In addition to the interpreter, IDLE has another part called the *file editor*. To open it, click the **File** menu at the top of the interactive shell. Then select **New File** if you are using OS X. A blank window will appear for you to type your program's code into, as shown in Figure 2-1.



```
Python 3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 20:20:57) [MSC v.1600 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

Untitled
File Edit Format Run Options Window Help
Ln: 1 Col: 0
Python 3.4.4 Shell
File Edit Shell Debug Options Window Help
Ln: 3 Col: 4

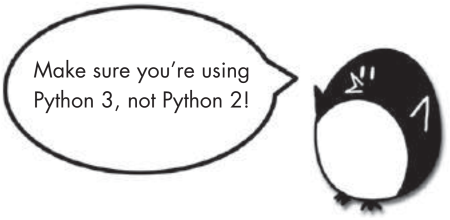
Figure 2-1: The file editor (left) and the interactive shell (right)

The two windows look similar, but just remember this: the interactive shell will have the >>> prompt. The file editor will not.

Creating the Hello World Program

It's traditional for programmers to make their first program display `Hello world!` on the screen. You'll create your own Hello World program now.

When you enter your program, remember not to enter the numbers at the left side of the code. They're there so this book can refer to the code by line number. The bottom-right corner of the file editor will tell you where the blinking cursor is so you can check which line of code you are on. Figure 2-2 shows that the cursor is on line 1 and column 0.



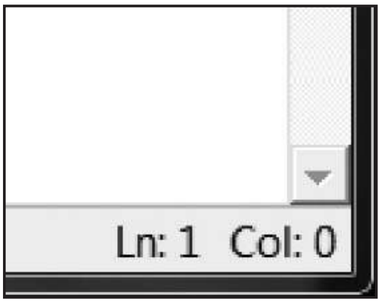


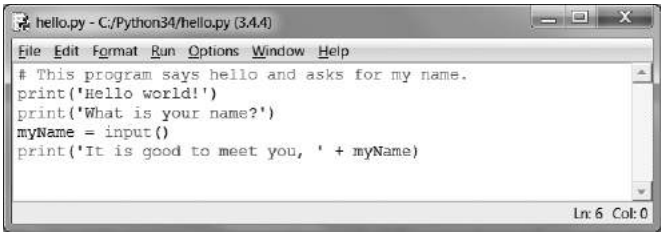
Figure 2-2: The bottom-right of the file editor tells you what line the cursor is on.

Enter the following text into the new file editor window. This is the program's *source code*. It contains the instructions Python will follow when the program is run.

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)
```

IDLE will write different types of instructions with different colors. After you're done typing the code, the window should look like Figure 2-3.


hello.py - C:/Python34/hello.py (3.4.4)
File Edit Format Run Options Window Help
Ln: 6 Col: 0

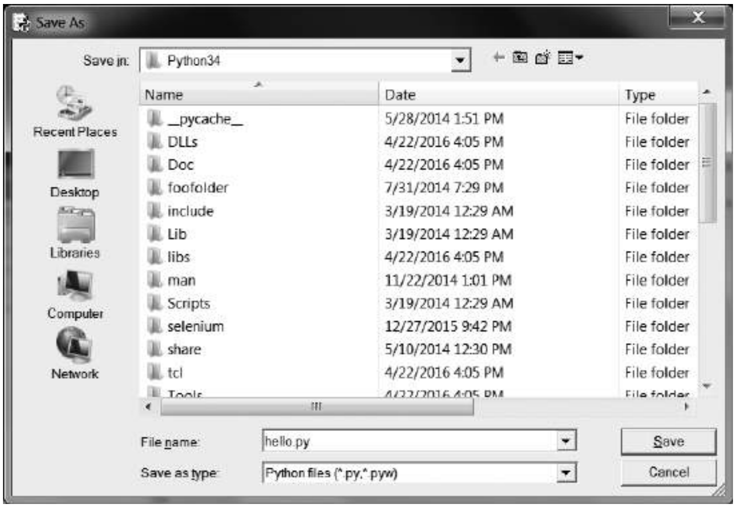
```
# This program says hello and asks for my name.
print('Hello world!')
print('What is your name?')
myName = input()
print('It is good to meet you, ' + myName)
```

Figure 2-3: The file editor will look like this after you enter your code.

Check to make sure your IDLE window looks the same.

Saving Your Program

Once you've entered your source code, save it by clicking **File ▸ Save As**. Or press CTRL-S to save with a keyboard shortcut. Figure 2-4 shows the Save As window that will open. Enter *hello.py* in the File name text field and then click **Save**.



Name	Date	Type
__pycache__	5/28/2014 1:51 PM	File folder
DLLs	4/22/2016 4:05 PM	File folder
Doc	4/22/2016 4:05 PM	File folder
foofolder	7/31/2014 7:29 PM	File folder
include	3/19/2014 12:29 AM	File folder
Lib	3/19/2014 12:29 AM	File folder
libs	4/22/2016 4:05 PM	File folder
man	11/22/2014 1:01 PM	File folder
Scripts	3/19/2014 12:29 AM	File folder
selenium	12/27/2015 9:42 PM	File folder
share	5/10/2014 12:30 PM	File folder
tcl	4/22/2016 4:05 PM	File folder
Tools	4/22/2016 4:05 PM	File folder

File name: hello.py
Save as type: Python files (*.py,*.pyw)

Figure 2-4: Saving the program

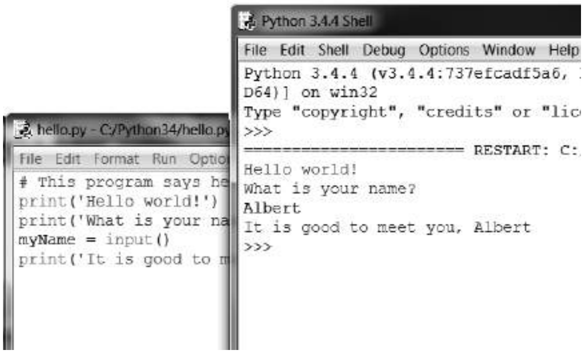
You should save your programs often while you write them. That way, if the computer crashes or you accidentally exit from IDLE, you won't lose much work.

Opening the Programs You've Saved

To load your previously saved program, click **File ▸ Open**. Select the *hello.py* file in the window that appears and click the **Open** button. Your saved *hello.py* program will open in the file editor.

Now it's time to run the program. Click **File ▸ Run Module**. Or just press **F5** from the file editor (FN-5 on OS X). Your program will run in the interactive shell.

Enter your name when the program asks for it. This will look like Figure 2-5.


hello.py - C:/Python34/hello.py
File Edit Format Run Optio
Python 3.4.4 Shell
File Edit Shell Debug Options Window Help

```
# This program says he
print('Hello world!')
print('What is your na
myName = input()
print('It is good to m
```

```
Python 3.4.4 (v3.4.4:737efcadf5a6,
D64)] on win32
Type "copyright", "credits" or "lic
>>>
======================= RESTART: C:
Hello world!
What is your name?
Albert
It is good to meet you, Albert
>>>
```

Figure 2-5: The interactive shell after you run hello.py

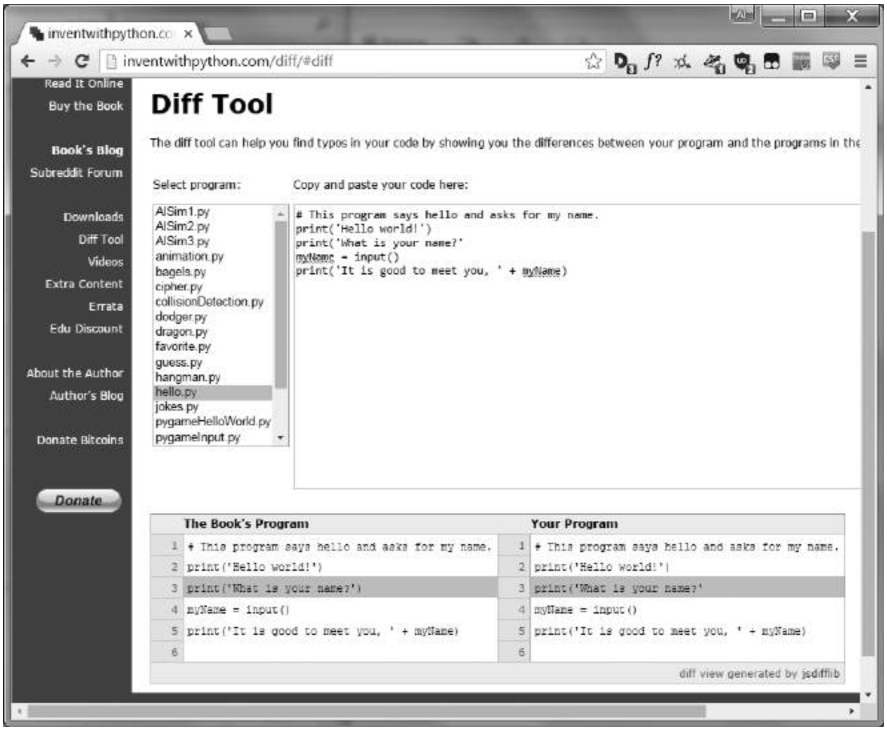
When you type your name and press ENTER, the program will greet you by name. Congratulations! You have written your first program and are now a computer programmer. Press **F5** again to run the program a second time and enter another name.

If you got an error, compare your code to this book's code with the online diff tool at *http://invpy.com/diff/*. Copy and paste your code from the file editor into the web page and click the **Compare** button. This tool will highlight any differences between your code and the code in this book, as shown in Figure 2-6.

While coding, if you get a `NameError` that looks like this:

```
Hello world!
What is your name?
Albert
Traceback (most recent call last):
  File “C:/Python26/test1.py”, line 4, in <module>
    myName = input()
  File “<string>”, line 1, in <module>
NameError: name ‘Albert’ is not defined
```

That means you are using Python 2 instead of Python 3. Install a version of Python 3 from *https://python.org/download/*. Rerun the program with Python 3.



Diff Tool

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

Select program:

AISim1.py
AISim2.py
AISim3.py
animation.py
bagels.py
cipher.py
collisionDetection.py
dodger.py
dragon.py
favorite.py
guess.py
hangman.py
hello.py
jokes.py
pygameHelloWorld.py
pygameInput.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)

diff view generated by jsdifflib

Figure 2-6: The diff tool at http://invpy.com/diff/

How the Hello World Program Works

Each line of code is an instruction interpreted by Python. These instructions make up the program. A computer program's instructions are like the steps in a recipe. Python completes each instruction in order, beginning from the top of the program and moving downward.

The step where Python is currently working in the program is called the *execution*. When the program starts, the execution is at the first instruction. After executing the instruction, Python moves down to the next instruction.

Let's look at each line of code to see what it's doing. We'll begin with line number 1.

Comments

The first line of the Hello World program is a *comment*.

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

Any text following a hash mark (`#`) is a comment. Comments are the programmer's notes about what the code does; they are not written for Python, but for you, the programmer. Python ignores comments when it runs a program. Programmers usually put a comment at the top of their code to give their program a title. The comment in the Hello World program tells you that the program says hello and asks you your name.

Functions

A *function* is kind of like a mini-program inside your program that contains several instructions for Python to execute. The great thing about functions is that you only need to know what they do, not how they do it. Python provides some built-in functions already. We use `print()` and `input()` in the Hello World program.

A *function call* is an instruction that tells Python to run the code inside a function. For example, your program calls the `print()` function to display a string on the screen. The `print()` function takes the string you type between the parentheses as input and displays that text on the screen.

The print() Function

Lines 2 and 3 of the Hello World program are calls to `print()`:

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

A value between the parentheses in a function call is an *argument*. The argument on line 2's `print()` function call is `'Hello world!'`, and the argument on line 3's `print()` function call is `'What is your name?'`. This is called *passing* the argument to the function.

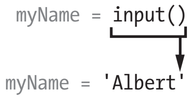
The input() Function

Line 4 is an assignment statement with a variable, `myName`, and a function call, `input()`.

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

When `input()` is called, the program waits for the user to enter text. The text string that the user enters becomes the value that the function call evaluates to. Function calls can be used in expressions anywhere a value can be used.

The value that the function call evaluates to is called the *return value*. (In fact, "the value a function call returns" means the same thing as "the value a function call evaluates to.") In this case, the return value of the `input()` function is the string that the user entered: their name. If the user enters `Albert`, the `input()` function call evaluates to the string `'Albert'`. The evaluation looks like this:



```
myName = input()

myName = 'Albert'
```

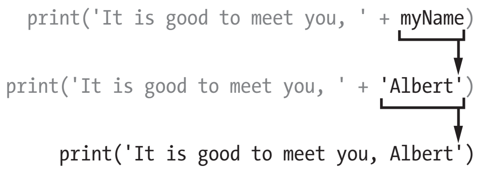
This is how the string value `'Albert'` gets stored in the `myName` variable.

Expressions in Function Calls

The last line in the Hello World program is another `print()` function call.

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

The expression `'It is good to meet you, ' + myName` is between the parentheses of `print()`. Because arguments are always single values, Python will first evaluate this expression and then pass that value as the argument. If `'Albert'` is stored in `myName`, the evaluation looks like this:



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

print('It is good to meet you, ' + 'Albert')

print('It is good to meet you, Albert')
```

This is how the program greets the user by name.

The End of the Program

Once the program executes the last line, it *terminates* or *exits*. This means the program stops running. Python forgets all of the values stored in variables, including the string stored in `myName`. If you run the program again and enter a different name, the program will think that is your name.

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

Remember, the computer does exactly what you program it to do. Computers are dumb and just follow the instructions you give them exactly. The computer doesn't care if you type in your name, someone else's name, or something silly. Type in anything you want. The computer will treat it the same way:

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

Variable Names

Giving variables descriptive names makes it easier to understand what a program does. You could have called the `myName` variable `abrahamLincoln` or `nAmE`, and Python would have run the program just the same. But those names don't really tell you much about what information the variable might hold. As Chapter 2 discussed, if you were moving to a new house and you labeled every moving box *Stuff*, that wouldn't be helpful at all! This book's interactive shell examples use variable names like `spam`, `eggs`, `ham`, and `bacon` because the variable names in these examples don't matter. However, this book's programs all use descriptive names, and so should your programs.

Variable names are *case-sensitive*, which means the same variable name in a different case is considered a different variable. So `spam`, `SPAM`, `Spam`, and `sPAM` are four different variables in Python. They each contain their own separate values. It's a bad idea to have differently cased variables in your program. Use descriptive names for your variables instead.

Variable names are usually lowercase. If there's more than one word in the variable name, it's a good idea to capitalize each word after the first. For example, the variable name `whatIHadForBreakfastThisMorning` is much easier to read than `whatihadforbreakfastthismorning`. Capitalizing your variables this way is called *camel case* (because it resembles the humps on a camel's back), and it makes your code more readable. Programmers also prefer using shorter variable names to make code easier to understand: `breakfast` or `foodThisMorning` is more readable than `whatIHadForBreakfastThisMorning`. These are *conventions*—optional but standard ways of doing things in Python programming.

Summary

Once you understand how to use strings and functions, you can start making programs that interact with users. This is important because text is the main way the user and the computer will communicate with each other. The user enters text through the keyboard with the `input()` function, and the computer will display text on the screen with the `print()` function.

Strings are just values of a new data type. All values have a data type, and the data type of a value will affect how the `+` operator functions.

Functions are used to carry out complicated instructions in your program. Python has many built-in functions that you'll learn about in this book. Function calls can be used in expressions anywhere a value is used.

The instruction or step in your program where Python is currently working is called the execution. In Chapter 3, you'll learn more about making the execution move in ways other than just straight down the program. Once you learn this, you'll be ready to create games!

3
GUESS THE NUMBER

In this chapter, you're going to make a Guess the Number game. The computer will think of a secret number from 1 to 20 and ask the user to guess it. After each guess the computer will tell the user if the number is too high or too low. The user wins if they can guess the number within six tries.

This is a good game to code because it covers many programming concepts in a short program. You'll learn how to convert values to different data types, and when you would need to do this. Since this program is a game, from now on we'll call the user the *player*.

TOPICS COVERED IN THIS CHAPTER

- `import` statements
- Modules
- The `randint()` function
- `for` statements
- Conditions
- Blocks
- The `str()`, `int()`, and `float()` functions
- Booleans
- Comparison operators
- The difference between `=` and `==`
- `if` statements
- The `break` keyword

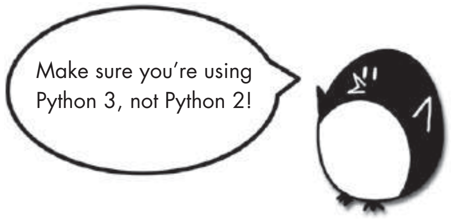
Sample Run of Guess the Number

Here's what the Guess the Number program looks like to the player when it's run. The player's input is marked in bold.

```
Hello! What is your name?
Albert
Well, Albert, I am thinking of a number between 1 and 20.
Take a guess.
10
Your guess is too high.
Take a guess.
2
Your guess is too low.
Take a guess.
4
Good job, Albert! You guessed my number in 3 guesses!
```

Source Code for Guess the Number

Open a new file editor window by clicking **File ▸ New File**. In the blank window that appears, enter the source code and save it as *guess.py*. Then run the program



by pressing **F5**. When you enter this code into the file editor, be sure to pay attention to the spacing at the front of the lines. Some lines need to be indented four or even eight spaces.

If you get errors after entering this code, compare the code you typed to the book's code with the online diff tool at *http://invpy.com/diff/guess/*.

guess.py

```
# This is a Guess the Number game.
import random

guessesTaken = 0

print('Hello! What is your name?')
myName = input()

number = random.randint(1, 20)
print('Well, ' + myName + ', I am thinking of a number between 1 and 20.')

for i in range(6):
    print('Take a guess.') # four spaces in front of "print"
    guess = input()
    guess = int(guess)

    if guess < number:
        print('Your guess is too low.') # eight spaces in front of "print"

    if guess > number:
        print('Your guess is too high.')

    if guess == number:
        break

if guess == number:
    guessesTaken = str(guessesTaken)
    print('Good job, ' + myName + '! You guessed my number in ' + guessesTaken + ' guesses!')

if guess != number:
    number = str(number)
    print('Nope. The number I was thinking of was ' + number + '.')
```

Importing the random Module

Let's take a look at the first two lines of this program.

```
# This is a Guess the Number game.
import random
```

The first line is a comment, which you saw in Chapter 2. Remember that Python will ignore everything after the # character. The comment here just reminds us what this program does.

The second line is an *`import` statement*. Remember, statements are instructions that perform some action but don't evaluate to a value like expressions do. You've already seen the assignment statement, which store a value in a variable.

While Python includes many built-in functions, some functions are written in separate programs called *modules*. You can use these functions by importing their modules into your program with an `import` statement.

Line 2 imports the `random` module so that the program can call the `randint()` function. This function will come up with a random number for the player to guess.

Now that you've imported the `random` module, you need to set up some variables to store values your program will use later.

Line 4 creates a new variable named `guessesTaken`.

```
4. guessesTaken = 0
```

You'll store the number of guesses the player has made in this variable. Since the player hasn't made any guesses at this point in the program, store the integer `0` here.

```
6. print('Hello! What is your name?')
7. myName = input()
```

Lines 6 and 7 are the same as the lines in the Hello World program in Chapter 2. Programmers often reuse code from their other programs to save themselves work.

Line 6 is a function call to `print()`. Remember that a function is like a mini-program inside your program. When your program calls a function, it runs this mini-program. The code inside `print()` displays the string argument you passed it on the screen.

Line 7 lets the player enter their name and stores it in the `myName` variable. Remember, the string might not really be the player's name; it's just whatever string the player typed. Computers are dumb and just follow their instructions no matter what.

Generating Random Numbers with the random.randint() Function

Now that your other variables are set up, you can use the `random` module's function to set the computer's secret number.

```
9. number = random.randint(1, 20)
```

Line 9 calls a new function named `randint()` and stores the return value in `number`. Remember, function calls can be part of expressions because they evaluate to a value.

The randint() function is provided by the random module, so you must call it with random.randint() (don't forget the period!) to tell Python that the function randint() is in the random module.

randint() will return a random integer between (and including) the two integer arguments you pass it. Line 9 passes 1 and 20, separated by commas, between the parentheses that follow the function name. The random integer that randint() returns is stored in a variable named number—this is the secret number the player is trying to guess.

Just for a moment, go back to the interactive shell and enter **import random** to import the random module. Then enter **random.randint(1, 20)** to see what the function call evaluates to. It will return an integer between 1 and 20. Repeat the code again and the function call will return a different integer. The randint() function returns a random integer each time, just as rolling a die will result in a random number each time. For example, enter the following into the interactive shell. The results you get when you call the randint() function will probably be different (it is random, after all!).

```
>>> import random
>>> random.randint(1, 20)
12
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
3
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
7
```

You can also try different ranges of numbers by changing the arguments. For example, enter **random.randint(1, 4)** to get only integers between 1 and 4 (including both 1 and 4). Or try **random.randint(1000, 2000)** to get integers between 1000 and 2000.

Enter this code in the interactive shell and see what numbers you get:

```
>>> random.randint(1, 4)
3
>>> random.randint(1000, 2000)
1294
```

You can change the game's code slightly to make the game behave differently. Try changing lines 9 and 10 from these lines:

```
 9. number = random.randint(1, 20)
10. print('Well, ' + name + ', I am thinking of a number between 1 and 20.')
```

. . . to these lines:

```
 9. number = random.randint(1, 100)
10. print('Well, ' + name + ', I am thinking of a number between 1 and 100.')
```

Now the computer will think of an integer between `1` and `100` instead of `1` and `20`. Changing line 9 will change the range of the random number, but remember to also change line 10 so that the game tells the player the new range instead of the old one.

You can use the `randint()` function whenever you want to add randomness to your games. You'll use randomness in many games. (Think of how many board games use dice.)

Welcoming the Player

After the computer assigns `number` a random integer, it greets the player:

```
10. print('Well, ' + myName + ', I am thinking of a number between 1 and 20.')
```

On line 10 `print()` welcomes the player by name, and tells them that the computer is thinking of a random number.

At first glance, it may look like there's more than one string argument in line 10, but examine the line carefully. The `+` operators between the three strings concatenate them into one string. And that one string is the argument passed to `print()`. If you look closely, you'll see that the commas are inside the quotes and part of the strings themselves.

Flow Control Statements

In previous chapters, the program execution started at the top instruction in the program and moved straight down, executing each instruction in order. But with the `for`, `if`, `else`, and `break` statements, you can make the execution loop or skip instructions based on conditions. These kinds of statements are *flow control statements*, since they change the flow of the program execution as it moves around your program.

Using Loops to Repeat Code

Line 12 is a `for` statement, which indicates the beginning of a `for` loop.

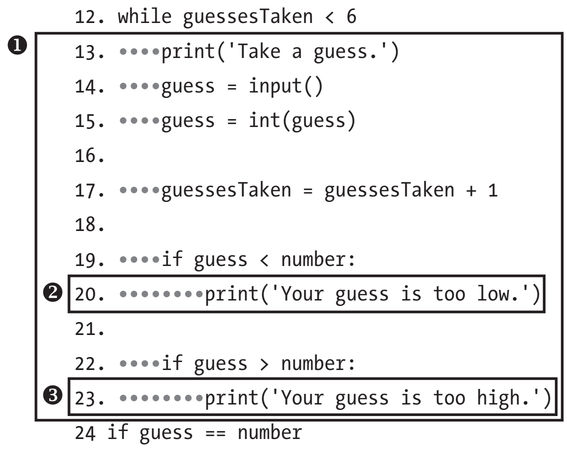
```
12. for i in range(6):
```

Loops let you execute code over and over again. Line 12 will repeat its code six times. A `for` statement begins with the `for` keyword, followed by a new variable name, the `in` keyword, a call to the `range()` function that specifies the number of loops it should do, and a colon. Let's go over a few additional concepts so that you can work with loops.

Grouping with Blocks

Several lines of code can be grouped together in a *block*. Every line in a block of code begins with at least the number of spaces as the first line in the block. You can tell where a block begins and ends by looking at the number of spaces at the front of the lines. This is the line's *indentation*.

Python programmers typically use four *additional* spaces of indentation to begin a block. Any following line that's indented by that same amount is part of the block. The block ends when there's a line of code with the *same indentation as before* the block started. There can also be blocks within other blocks. Figure 3-1 shows a code diagram with the blocks outlined and numbered.



```
12. while guessesTaken < 6
13. ••••print('Take a guess.')
14. ••••guess = input()
15. ••••guess = int(guess)
16.
17. ••••guessesTaken = guessesTaken + 1
18.
19. ••••if guess < number:
20. ••••••••print('Your guess is too low.')
21.
22. ••••if guess > number:
23. ••••••••print('Your guess is too high.')
24 if guess == number
```

Figure 3-1: An example of blocks and their indentation. The black dots represent spaces.

In Figure 3-1, line 12 has no indentation and isn't inside any block. Line 13 has an indentation of four spaces. Since this line is indented more than the previous line, a new block starts here. Every line following this one with the same amount of indentation or more is considered part of block ❶. If Python encounters another line with less indentation than the block's first line, the block has ended. Blank lines are ignored.

Line 20 has an indentation of eight spaces, which starts block ❷ is *inside* the first block.

Line 22 only has four spaces. Because the indentation has decreased, you know that line 20's block has ended. Line 20 is the only line in block ❷. Line 22 has four spaces, so you know it's in block ❶.

Line 23 increases the indentation to eight spaces, so another new block within a block has started: block ❸.

Looping with for Statements

The `for` statement marks the beginning of a loop. Loops execute the same code repeatedly. When the execution reaches a `for` statement, it enters the block that follows the `for` statement. After running all the code in this block, the execution moves back to the top of the block to run the code all over again.

Enter the following into the interactive shell:

```
>>> for i in range(3):
    print('Hello! i is set to', i)


Hello! i is set to 0
Hello! i is set to 1
Hello! i is set to 2
```

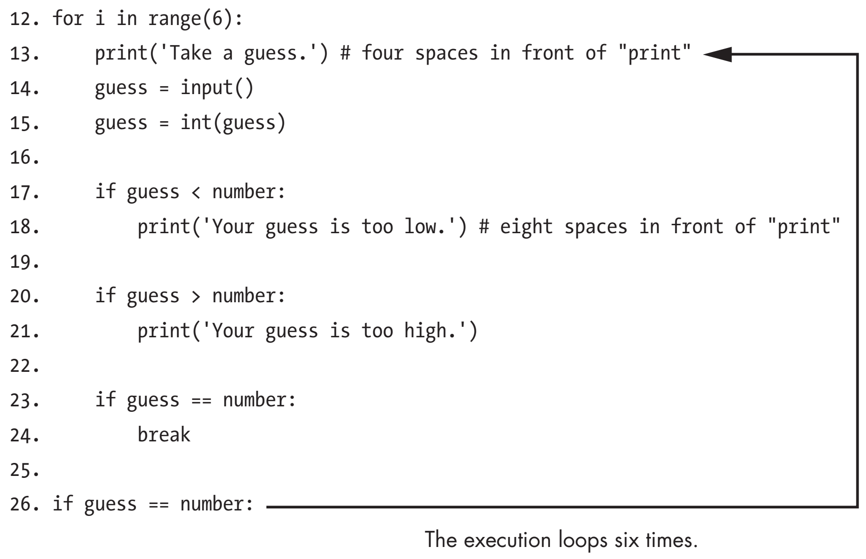
Notice that after you typed `for i in range(3):` and pressed ENTER, the interactive shell didn't show another `>>>` prompt because it was expecting you to type a block of code. After you type `print('Hello! i is set to', i')` and press ENTER, you have to press ENTER again to insert a blank line. This blank line tells the interactive shell you are done with the block. (This applies only to the interactive shell. When writing *.py* files in the file editor, you don't need to insert a blank line.)

Let's look at the `for` loop on line 12:

```
12. for i in range(6):
13.     print('Take a guess.') # four spaces in front of "print"
14.     guess = input()
15.     guess = int(guess)
16.
17.     if guess < number:
18.         print('Your guess is too low.') # eight spaces in front of "print"
19.
20.     if guess > number:
21.         print('Your guess is too high.')
22.
23.     if guess == number:
24.         break
25.
26. if guess == number:
```

In Guess the Number, the `for` block begins at the `for` statement on line 12, and the first line after the `for` block is line 26.

A `for` statement always has a colon (`:`) after the condition. Statements that end with a colon expect a new block on the next line. This is illustrated in Figure 3-2.



```
12. for i in range(6):
13.     print('Take a guess.') # four spaces in front of "print"
14.     guess = input()
15.     guess = int(guess)
16.
17.     if guess < number:
18.         print('Your guess is too low.') # eight spaces in front of "print"
19.
20.     if guess > number:
21.         print('Your guess is too high.')
22.
23.     if guess == number:
24.         break
25.
26. if guess == number:
```

Figure 3-2: The for loop's condition

Figure 3-2 shows how the execution flows. The execution will enter the `for` block at line 13 and keep going down. Once the program reaches the end of the `for` block, instead of going down to the next line, the execution loops back up to the start of the `for` block at line 13. It does this six times because of the `range(6)` function call in the `for` statement. Each time the execution goes through the loop is called an *iteration*.

Think of the `for` statement as saying, "Execute the code in the following block a certain number of times."

Getting the Player's Guess

Lines 13 and 14 ask the player to guess what the secret number is and lets them enter their guess:

```
13.     print('Take a guess.') # Four spaces in front of "print"
14.     guess = input()
```

That number the player enters is stored in a variable named `guess`.

Converting Values with the int(), float(), and str() Functions

Line 15 calls a new function called `int()`.

```
15.     guess = int(guess)
```

The `int()` function takes one argument and returns the argument's value as an integer.

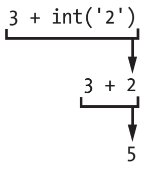
Enter the following into the interactive shell to see how the `int()` function works:

```
>>> int('42')
42
```

The `int('42')` call will return the integer value `42`.

```
>>> 3 + int('2')
5
```

The `3 + int('2')` line shows an expression that uses the return value of `int()` as part of an expression. It evaluates to the integer value `5`:



Even though you can pass a string to `int()`, you cannot pass it just any string. Passing `'forty-two'` to `int()` will result in an error.:

```
>>> int('forty-two')
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
int('forty-two')
ValueError: invalid literal for int() with base 10: 'forty-two'
```

The string you pass to `int()` must be made of numbers.

In Guess the Number, we get the players number using the `input()` function. Remember, the `input()` function always returns a *string* of text the player entered. If the player types `5`, the `input()` function will return the string value `'5'`, not the integer value `5`. But we'll need to compare the player's number with an integer later, and Python cannot use the `<` and `>` comparison operators to compare a string and an integer value:

```
>>> 4 < '5'
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    4 < '5'
TypeError: unorderable types: int() < str()
```

We need to convert the string into an integer.

```
    guess = input()
    guess = int(guess)
```

On line 14, we assign the guess variable to the string value of whatever number the player typed. Line 15 overwrites the string value in guess with the integer value returned by int().The code int(guess) returns a new integer value based on the string it was provided, and guess = assigns that new value to guess. This lets the code later in the program compare whether guess is greater than, less than, or equal to the secret number in the number variable.

The float() and str() functions will similarly return float and string versions of the arguments passed to them. Enter the following into the interactive shell:

```
>>> float('42')
42.0
>>> float(42)
42.0
```

When the string '42' or the integer 42 is passed to float(), the float 42.0 is returned.

Now try using the str() function:

```
>>> str(42)
'42'
>>> str(42.0)
'42.0'
```

When the integer 42 is passed to str(), the string '42' is returned. But when the float 42.0 is passed to str(), the string '42.0' is returned.

Using the int(), float(), and str() functions, you can take a value of one data type and return it as a value of a different data type.

The Boolean Data Type

Every value in Python belongs to one data type. The data types that have been introduced so far are integers, floats, strings, and now Booleans. The *Boolean* data type has only two values: True or False. Boolean values must be entered with a capital T or F and the rest of the value's name in lowercase.

Boolean values can be stored in variables just like the other data types:

```
>>> spam = True
>>> eggs = False
```

In this example, you set spam to True and eggs to False. Remember not to type true or false with all lowercase letters.

You will use Boolean values (called *bools* for short) with comparison operators to form conditions. We'll cover comparison operators first and then go over conditions.

Comparison Operators

Comparison operators compare two values and evaluate to a `True` or `False` Boolean value. Table 3-1 lists all of the comparison operators.

Table 3-1: Comparison Operators

Operator sign	Operator name
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`==`	Equal to
`!=`	Not equal to

You've already read about the `+`, `-`, `*`, and `/` math operators. Like any operator, comparison operators combine with values to form expressions such as `guessesTaken < 6`.

Line 17 of the Guess the Number program uses the less than comparison operator:

```
17.     if guess < number:
```

We'll discuss `if` statements in more detail shortly; for now, let's just look at the expression that follows the `if` keyword (the `guess < number` part). This expression contains two values (the value in the variables `guess` and `number`) connected by an operator (the `<`, or less than sign).

Checking for True or False with Conditions

A *condition* is an expression that combines two values with a comparison operator (such as `<` or `>`) and evaluates to a Boolean value. A condition is just another name for an expression that evaluates to `True` or `False`. One place we use conditions is in `if` statements.

For example, the condition `guess < number` on line 17 asks, "Is the value stored in `guessesTaken` less than the value stored in `number`?" If so, then the condition evaluates to `True`. If not, the condition evaluates to `False`.

Say that `guess` stores the integer `10` and `number` stores the integer `16`. Because `10` is less than `16`, this condition evaluates to the Boolean value of `True`. The evaluation would look like this:

```
guess < number
             ↓
       10 < 16
             ↓
          True
```

Experimenting with Booleans, Comparison Operators, and Conditions

Enter the following expressions in the interactive shell to see their Boolean results:

```
>>> 0 < 6
True
>>> 6 < 0
False
```

The condition `0 < 6` returns the Boolean value `True` because the number `0` is less than the number `6`. But because `6` isn't less than `0`, the condition `6 < 0` evaluates to `False`.

Notice that `10 < 10` evaluates to `False` because the number `10` isn't smaller than the number `10`:

```
>>> 10 < 10
False
```

The values are the same. If Alice were the same height as Bob, you wouldn't say that Alice is taller than Bob or that Alice is shorter than Bob. Both of those statements would be false.

Now enter these expressions into the interactive shell:

```
>>> 10 == 10
True
>>> 10 == 11
False
>>> 11 == 10
False
>>> 10 != 10
False
```

In this example, `10` is equal to `10`, so `10 == 10` evaluates to `True`. But `10` is not equal to `11`, so `10 == 11` is `False`. Even if the order is flipped, `11` is still not equal to `10`, so `11 == 10` is `False`. Finally, `10` is equal to `10`, so `10 != 10` is `False`.

You can also evaluate string expressions with comparison operators:

```
>>> 'Hello' == 'Hello'
True
>>> 'Goodbye' != 'Hello'
True
>>> 'Hello' == 'HELLO'
False
```

`'Hello'` is equal to `'Hello'`, so `'Hello' == 'Hello'` is `True`. `'Goodbye'` is not equal to `'Hello'`, so `'Goodbye' != 'Hello'` is also `True`.

Notice that the last line evaluates to `False`. Capital and lowercase letters are not the same in Python, so `'Hello'` is not equal to `'HELLO'`.

String and integer values will never be equal to each other. For example, enter the following into the interactive shell:

```
>>> 42 == 'Hello'
False
>>> 42 != '42'
True
```

In the first example, `42` is an integer and `'Hello'` is a string, so the values are not equal and the expression evaluates to `False`. In the second example, the string `'42'` is still not an integer, so the expression "the integer `42` is not equal to the string `'42'`" evaluates to `True`.

The Difference Between = and ==

Try not to confuse the assignment operator `=` and the equal to comparison operator `==`. The equal sign, `=`, is used in assignment statements to store a value to a variable, whereas the double equal sign, `==`, is used in expressions to see whether two values are equal. It's easy to accidentally use one when you mean to use the other.

It might help to remember that the equal to comparison operator `==` has two characters in it, just as the not equal to comparison operator `!=` has two characters in it.

The bool() Function

Like the `str()` and `int()` functions, the `bool()` function returns a Boolean value of the value passed to it. Enter the following into the interactive shell:

```
>>> bool('')
False
>>> bool('any nonempty string')
True
```

Python considers some non-Boolean values to be *truthy* or *falsey*. An empty string or the integer `0` are considered falsey while all other strings and integers are truthy. When an empty string is passed to `bool()`, the Boolean `False` is returned. When any nonempty string is passed to `bool()`, the Boolean `True` is returned.

if Statements

Line 17 is an `if` statement:

```
17.     if guess < number:
18.         print('Your guess is too low.') # eight spaces in front of
"print"
```

The code block following the if statement will run if the if statement's condition evaluates to True. If the condition is False, the code in the if block is skipped. Using if statements, you can make the program run certain code only when you want it to.

Line 17 checks whether the player's guess is less than the computer's secret number. If so, then the execution moves inside the if block on line 18 and prints a message telling the player their guess was too low.

Line 20 checks whether the player's guess is greater than the secret number:

```
20.     if guess > number:
21.         print('Your guess is too high.')
```

If this condition is True, then the print() function call tells the player that their guess is too high.

Leaving Loops Early with the break Statement

The if statement on line 23 checks if the number the player guessed is equal to the secret number. If it is, the program runs the break statement on line 24.

```
23.     if guess == number:
24.         break
```

A *break statement* tells the execution to jump immediately out of the for block to the first line after the end of the for block. The break statement is found only inside loops, such as in a for block.

Checking If the Player Won

The for block ends at the next line of code with no indentation, which is line 26:

```
26. if guess == number:
```

The execution leaves the for block either because it has looped six times (when the player runs out of guesses) or because the break statement on line 24 has executed (when the player guesses the number correctly).

Line 26 checks whether the player guessed correctly. If so, the execution enters the if block at line 27:

```
27.     guessesTaken = str(guessesTaken)
28.     print('Good job, ' + myName + '! You guessed my number in ' +
guessesTaken + ' guesses!')
```

Lines 27 and 28 execute only if the condition in the `if` statement on line 26 was `True` (that is, if the player correctly guessed the computer's number).

Line 27 calls the `str()` function, which returns the string form of `guessesTaken`. Line 28 concatenates strings to tell the player they have won and how many guesses it took. Only string values can concatenate to other strings. This is why line 27 had to change `guessesTaken` to the string form. Otherwise, trying to concatenate a string to an integer would cause Python to display an error.

Checking If the Player Lost

If the player runs out of guesses, the execution will go to this line of code:

```
30. if guess != number:
```

Line 30 uses the not equal to comparison operator `!=` to check if the player's last guess is not equal to the secret number. If this condition evaluates to `True`, the execution moves into the `if` block on line 31.

Lines 31 and 32 are inside the `if` block, and execute only if the condition on line 30 is `True`.

```
31.     number = str(number)
32.     print('Nope. The number I was thinking of was ' + number + '.')
```

In this block, the program tells the player what the secret number was. This requires concatenating strings, but `number` stores an integer value. Line 31 overwrites `number` with a string so that it can be concatenated to the `'Nope. The number I was thinking of was '` and `'.'` strings on line 32.

At this point, the execution has reached the end of the code, and the program terminates. Congratulations! You've just programmed your first real game!

You can adjust the game's difficulty by changing the number of guesses the player gets. To give the player only four guesses, change the code on line 12:

```
12. for i in range(6):
```

. . . to this line:

```
12. for i in range(4):
```

By passing `4` to `range()`, you ensure that the code inside the loop runs only four times instead of six. This makes the game much more difficult. To make the game easier, pass a larger integer to the `range()` function call. This will cause the loop to run a few *more* times and accept *more* guesses from the player.

Summary

Programming is just the action of writing code for programs—that is, creating programs that can be executed by a computer.

When you see someone using a computer program (for example, playing your Guess the Number game), all you see is some text appearing on the screen. The program decides what exact text to display on the screen (the program's *output*) based on its instructions and on the text that the player typed on the keyboard (the program's *input*). A program is just a collection of instructions that act on the user's input.

There are a few different kinds of instructions:

- **Expressions** are values connected by operators. Expressions are all evaluated down to a single value. For example, `2 + 2` evaluates to `4` or `'Hello' + ' ' + 'World'` evaluates to `'Hello World'`. When expressions are next to the `if` and `for` keywords, you can also call them *conditions*.
- **Assignment statements** store values in variables so you can remember the values later in the program.
- **The `if`, `for`, and `break` statements** are flow control statements that can make the execution skip instructions, loop over instructions, or break out of loops. Function calls also change the flow of execution by jumping to the instructions inside of a function.
- **The `print()` and `input()` functions** display text on the screen and get text from the keyboard. Instructions that deal with the *input* and *output* of the program are called *I/O* (pronounced "eye oh").

That's it—just those four things. Of course, there are many details about those four types of instructions. In later chapters, you'll learn about more data types and operators, more flow control statements, and many other functions that come with Python. There are also different types of I/O beyond text, such as input from the mouse and sound and graphics output.

4

JOKES

This chapter's program tells a few jokes to the user, and demonstrates more advanced ways to use strings with the `print()` function.

TOPICS COVERED IN THIS CHAPTER

- Escape characters
- Using single quotes and double quotes for strings
- Using `print()`'s `end` keyword parameter to skip newlines

Making the Most of print()

Most of the games in this book will have simple text for input and output. The input is typed on the keyboard by the user, and the output is the text displayed on the screen. You've already learned how to display simple text output with the `print()` function. Not let's take a deeper look at how strings and `print()` work in Python.

Sample Run of Jokes

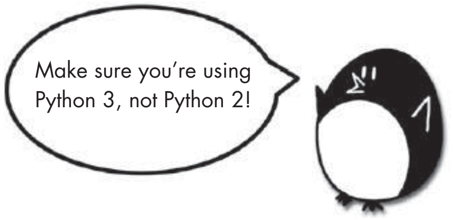
Here's what the user sees when they run the Jokes program:

```
What do you get when you cross a snowman with a vampire?
Frostbite!
What do dentists call an astronaut's cavity?
A black hole!
Knock knock.
Who's there?
Interrupting cow.
Interrupting cow wh-MOO!
```

Source Code of Jokes

Open a new file editor window by clicking **File ▸ New Window**. In the blank window that appears, enter the source code and save it as *jokes.py*. Then run the program by pressing **F5**.

If you get errors after entering this code, compare the code you typed to the book's code with the online diff tool at *http://invpy.com/diff/jokes/*.



jokes.py

```
 1. print('What do you get when you cross a snowman with a vampire?')
 2. input()
 3. print('Frostbite!')
 4. print()
 5. print('What do dentists call an astronaut\'s cavity?')
 6. input()
 7. print('A black hole!')
 8. print()
 9. print('Knock knock.')
10. input()
11. print("Who's there?")
12. input()
13. print('Interrupting cow.')
```

```
14. input()
15. print('Interrupting cow wh', end='')
16. print('-MOO!')
```

How the Code Works

Let's start by looking at the first four lines of code:

```
 1. print('What do you get when you cross a snowman with a vampire?')
 2. input()
 3. print('Frostbite!')
 4. print()
```

Lines 1 and 3 use the `print()` function call to ask and give the answer to the first joke. You don't want the user to immediately read the joke's punchline, so there's a call to the `input()` function after the first `print()` instance. The user will read the joke, press ENTER, and then read the punchline.

The user can still type in a string and press ENTER, but this returned string isn't being stored in any variable. The program will just forget about it and move to the next line of code.

The last `print()` function call doesn't have a string argument. This tells the program to just print a blank line. Blank lines are useful to keep the text from looking crowded.

Escape Characters

Lines 5–8 print the question and answer to the next joke:

```
 5. print('What do dentists call an astronaut\'s cavity?')
 6. input()
 7. print('A black hole!')
 8. print()
```

On line 5, there's a backslash right before the single quote: `\'`. (Note that `\` is a backslash, and `/` is a forward slash.) This backslash tells you that the letter right after it is an escape character. An *escape character* lets you print special characters that are difficult or impossible to enter into the source code, such as the single quote in a string value that begins and ends with single quotes.

In this case, if we didn't include the backslash, the single quote in `astronaut\'s` would be interpreted as the end of the string. But this quote needs to be *part of* the string. The escaped single quote tells Python that it should include the single quote in the string.

But what if you actually want to display a backslash?

Switch from your *jokes.py* program to the interactive shell and enter this print() statement:

```
>>> print('They flew away in a green\teal helicopter.')
They flew away in a green    eal helicopter.
```

This instruction wouldn't print a backslash because the t in teal was interpreted as an escape character since it came after a backslash. The \t simulates pressing TAB on your keyboard.

This line will give you the correct output:

```
>>> print('They flew away in a green\\teal helicopter.')
They flew away in a green\teal helicopter.
```

This way the \\ is a backslash character, and there is no \t to interpret as TAB.

Table 4-1 is a list of some escape characters in Python, including \n, which is the newline escape character that you have used before.

Table 4-1: Escape Characters

Escape character	What is actually printed
`\\`	Backslash (\)
`\'`	Single quote (')
`\"`	Double quote (")
`\n`	Newline
`\t`	TAB

There are a few more escape characters in Python, but these are the characters you will most likely need for creating games.

Single Quotes and Double Quotes

While we're still in the interactive shell, let's take a closer look at quotes. Strings don't always have to be between single quotes in Python. You can also put them between double quotes. These two lines print the same thing:

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

But you cannot mix quotes. This line will give you an error because it uses both quote types at once:

```
>>> print('Hello world")
SyntaxError: EOL while scanning single-quoted string
```

I like to use single quotes so I don't have to hold down SHIFT to type them. They're easier to type, and Python doesn't care either way.

Also, note that just as you need the `\'` to have a single quote in a string surrounded by single quotes, you need the `\"` to have a double quote in a string surrounded by double quotes. Look at this example:

```
>>> print('I asked to borrow Abe\'s car for a week. He said, "Sure."')
I asked to borrow Abe's car for a week. He said, "Sure."
```

You use single quotes to surround the string, so you need to add a backslash before the single quote in `Abe\'s`. But the double quotes in `"Sure."` don't need backslashes. The Python interpreter is smart enough to know that if a string starts with one type of quote, the other type of quote doesn't mean the string is ending.

Now check out another example:

```
>>> print("She said, \"I can't believe you let them borrow your car.\"")
She said, "I can't believe you let them borrow your car."
```

The string is surrounded in double quotes, so you need to add backslashes for all of the double quotes within the string. You don't need to escape the single quote in `can't`.

To summarize, in the single-quote strings you don't need to escape double quotes but do need to escape single quotes, and in the double-quote strings you don't need to escape single quotes but do need to escape double quotes.

print()'s end Keyword Parameter

Now let's go back to *jokes.py* and take a look at lines 9–12:

```
 9. print('Knock knock.')
10. input()
11. print("Who's there?")
12. input()
13. print('Interrupting cow.')
14. input()
15. print('Interrupting cow wh', end='')
16. print('-MOO!')
```

Did you notice the second argument in line 15's `print()` function? Normally, `print()` adds a newline character to the end of the string it prints. This is why a blank `print()` function will just print a newline. But `print()` can optionally have a second parameter: `end`.

An argument is the value passed in a function call. The blank string passed to `print()` is called a *keyword argument*. The `end` in `end=''` is called a keyword parameter. It has a specific name, and to pass a keyword argument for this keyword parameter you must type `end=` before it.

When we run this section of code, the output is:

```
Knock knock.
Who's there?
Interrupting cow.
Interrupting cow wh-MOO!
```

Because we passed a blank string to the `end` parameter, the `print()` function will add a blank string instead of adding a newline. This is why `'-MOO!'` appears next to the previous line, instead of on its own line. There was no newline after the `'Interrupting cow wh'` string was printed.

Summary

This chapter explores the different ways you can use the `print()` function. Escape characters are used for characters that are difficult to type into the code with the keyboard. If you want to use special characters in a string, you must use a backslash escape character, `\`, followed by another letter for the special character. For example, `\n` would be a newline. If your special character is a backslash itself, you would use `\\`.

`print()` automatically appends a newline character to the end of a string. Most of the time, this is a helpful shortcut. But sometimes you don't want a newline character. To change this, you can pass a blank string as the keyword argument for `print()`'s `end` keyword parameter. For example, to print `spam` to the screen without a newline character, you would call `print('spam', end='')`.