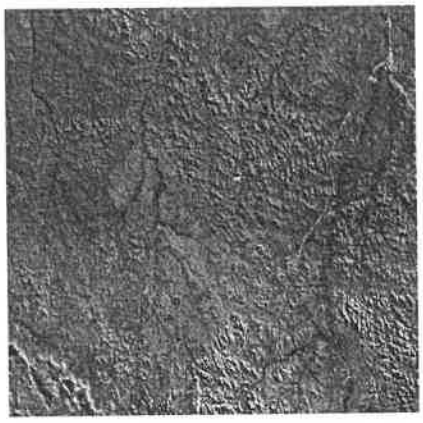
CHAPTER 1

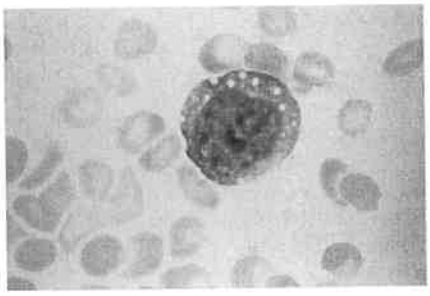
What's Programming?



(Photo credit: NASA/Goddard Space Flight Center Scientific Visualization Studio)

Take a look at the pictures above. The first one shows forest cover in the Amazon basin in 1975. The second one shows the same area twenty-six years later. Anyone can see that much of the rainforest has been destroyed, but how much is "much"?

Now look at this:



(Photo credit: CDC)

Are these blood cells healthy? Do any of them show signs of leukemia? It would take an expert doctor a few minutes to tell. Multiply those minutes by the number of people who need to be screened. There simply aren't enough human doctors in the world to check everyone.

This is where computers come in. Computer programs can measure the differences between two pictures and count the number of oddly shaped platelets in a blood sample. Geneticists use programs to analyze gene sequences; statisticians, to analyze the spread of diseases; geologists, to predict the effects of earthquakes; economists, to analyze fluctuations in the stock market; and climatologists, to study global warming. More and more scientists are writing programs to help them do their work. In turn, those programs are making entirely new kinds of science possible.

Of course, computers are good for a lot more than just science. We used computers to write this book. Your smartphone is a pretty powerful computer; you've probably used one today to chat with friends, check your lecture notes, or look for a restaurant that serves pizza *and* Chinese food. Every day, someone figures out how to make a computer do something that has never been done before. Together, those "somethings" are changing the world.

This book will teach you how to make computers do what *you* want them to do. You may be planning to be a doctor, a linguist, or a physicist rather than a full-time programmer, but whatever you do, being able to program is as important as being able to write a letter or do basic arithmetic.

We begin in this chapter by explaining what programs and programming are. We then define a few terms and present some useful bits of information for course instructors.

Programs and Programming

A *program* is a set of instructions. When you write down directions to your house for a friend, you are writing a program. Your friend "executes" that program by following each instruction in turn.

Every program is written in terms of a few basic operations that its reader already understands. For example, the set of operations that your friend can understand might include the following: "Turn left at Darwin Street," "Go forward three blocks," and "If you get to the gas station, turn around—you've gone too far."

Computers are similar but have a different set of operations. Some operations are mathematical, like "Take the square root of a number," whereas others include "Read a line from the file named `data.txt`" and "Make a pixel blue."

The most important difference between a computer and an old-fashioned calculator is that you can "teach" a computer new operations by defining them in terms of old ones. For example, you can teach the computer that "Take the average" means "Add up the numbers in a sequence and divide by the sequence's size." You can then use the operations you have just defined to create still more operations, each layered on top of the ones that came before. It's a lot like creating life by putting atoms together to make proteins and then combining proteins to build cells, combining cells to make organs, and combining organs to make a creature.

Defining new operations and combining them to do useful things is the heart and soul of programming. It is also a tremendously powerful way to think about other kinds of problems. As Professor Jeannette Wing wrote in *Computational Thinking [Win06]*, computational thinking is about the following:

- *Conceptualizing, not programming.* Computer science isn't computer programming. Thinking like a computer scientist means more than being able to program a computer: it requires thinking at multiple levels of abstraction.
- *A way that humans, not computers, think.* Computational thinking is a way humans solve problems; it isn't trying to get humans to think like computers. Computers are dull and boring; humans are clever and imaginative. We humans make computers exciting. Equipped with computing devices, we use our cleverness to tackle problems we wouldn't dare take on before the age of computing and build systems with functionality limited only by our imaginations.
- *For everyone, everywhere.* Computational thinking will be a reality when it becomes so integral to human endeavors it disappears as an explicit philosophy.

We hope that by the time you have finished reading this book, you will see the world in a slightly different way.

What's a Programming Language?

Directions to the nearest bus station can be given in English, Portuguese, Mandarin, Hindi, and many other languages. As long as the people you're talking to understand the language, they'll get to the bus station.

In the same way, there are many programming languages, and they all can add numbers, read information from files, and make user interfaces with windows and buttons and scroll bars. The instructions look different, but

they accomplish the same task. For example, in the Python programming language, here's how you add 3 and 4:

```
3 + 4
```

But here's how it's done in the Scheme programming language:

```
(+ 3 4)
```

They both express the same idea—they just look different.

Every programming language has a way to write mathematical expressions, repeat a list of instructions a number of times, choose which of two instructions to do based on the current information you have, and much more. In this book, you'll learn how to do these things in the Python programming language. Once you understand Python, learning the next programming language will be much easier.

What's a Bug?

Pretty much everyone has had a program crash. A standard story is that you were typing in a paper when, all of a sudden, your word processor crashed. You had forgotten to save, and you had to start all over again. Old versions of Microsoft Windows used to crash more often than they should have, showing the dreaded "blue screen of death." (Happily, they've gotten a *lot* better in the past several years.) Usually, your computer shows some kind of cryptic error message when a program crashes.

What happened in each case is that the people who wrote the program told the computer to do something it couldn't do: open a file that didn't exist, perhaps, or keep track of more information than the computer could handle, or maybe repeat a task with no way of stopping other than by rebooting the computer. (Programmers don't mean to make these kinds of mistakes, they are just part of the programming process.)

Worse, some bugs don't cause a crash; instead, they give incorrect information. (This is worse because at least with a crash you'll notice that there's a problem.) As a real-life example of this kind of bug, the calendar program that one of the authors uses contains an entry for a friend who was born in 1978. That friend, according to the calendar program, had his 5,875,542nd birthday this past February. Bugs can be entertaining, but they can also be tremendously frustrating.

Every piece of software that you can buy has bugs in it. Part of your job as a programmer is to minimize the number of bugs and to reduce their severity. In order to find a bug, you need to track down where you gave the wrong

instructions, then you need to figure out the right instructions, and then you need to update the program without introducing other bugs.

Every time you get a software update for a program, it is for one of two reasons: new features were added to a program or bugs were fixed. It's always a game of economics for the software company: are there few enough bugs, and are they minor enough or infrequent enough in order for people to pay for the software?

In this book, we'll show you some fundamental techniques for finding and fixing bugs and also show you how to prevent them in the first place.

The Difference Between Brackets, Braces, and Parentheses

One of the pieces of terminology that causes confusion is what to call certain characters. Several dictionaries use these names, so this book does too:

() Parentheses

[] Brackets

{} Braces (Some people call these *curly brackets* or *curly braces*, but we'll stick to just *braces*.)

Installing Python

Installation instructions and use of the IDLE programming environment are available on the book's website: http://pragprog.com/titles/gwpy3/practical-programming.

CHAPTER 2

Hello, Python

Programs are made up of commands that tell the computer what to do. These commands are called *statements*, which the computer executes. This chapter describes the simplest of Python's statements and shows how they can be used to do arithmetic, which is one of the most common tasks for computers and also a great place to start learning to program. It's also the basis of almost everything that follows.

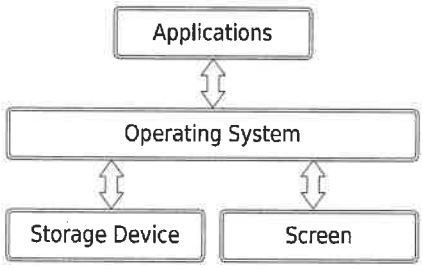
How Does a Computer Run a Python Program?

In order to understand what happens when you're programming, it helps to have have a mental model of how a computer executes a program.

The computer is assembled from pieces of hardware, including a *processor* that can execute instructions and do arithmetic, a place to store data such as a *hard drive*, and various other pieces, such as a screen, a keyboard, an Ethernet controller for connecting to a network, and so on.

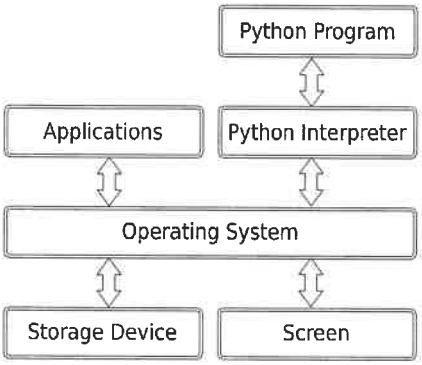
To deal with all these pieces, every computer runs some kind of *operating system*, such as Microsoft Windows, Linux, or macOS. An operating system, or OS, is a program; what makes it special is that it's the only program on the computer that's allowed direct access to the hardware. When any other application (such as your browser, a spreadsheet program, or a game) wants to draw on the screen, find out what key was just pressed on the keyboard, or fetch data from storage, it sends a request to the OS (see the top image on page 8).

This may seem like a roundabout way of doing things, but it means that only the people writing the OS have to worry about the differences between one graphics card and another and whether the computer is connected to a network through Ethernet or wireless. The rest of us—everyone analyzing



scientific data or creating 3D virtual chat rooms—only have to learn our way around the OS, and our programs will then run on thousands of different kinds of hardware.

Today, it's common to add another layer between the programmer and the computer's hardware. When you write a program in Python, Java, or Visual Basic, it doesn't run directly on top of the OS. Instead, another program, called an *interpreter* or *virtual machine*, takes your program and runs it for you, translating your commands into a language the OS understands. It's a lot easier, more secure, and more portable across operating systems than writing programs directly on top of the OS:



There are two ways to use the Python interpreter. One is to tell it to execute a Python program that is saved in a file with a .py extension. Another is to interact with it in a program called a *shell*, where you type statements one at a time. The interpreter will execute each statement when you type it, do what the statement says to do, and show any output as text, all in one window. We will explore Python in this chapter using a Python shell.

Install Python Now (If You Haven't Already)

If you haven't yet installed Python 3.6, please do so now. (Python 2 won't do; there are significant differences between Python 2 and Python 3, and this book uses Python 3.6.) Locate installation instructions on the book's website: http://pragprog.com/titles/gwpy3/practical-programming.

Programming requires practice: you won't learn how to program just by reading this book, much like you wouldn't learn how to play guitar just by reading a book on how to play guitar.

Python comes with a program called IDLE, which we use to write Python programs. IDLE has a Python shell that communicates with the Python interpreter and also allows you to write and run programs that are saved in a file.

We *strongly* recommend that you open IDLE and follow along with our examples. Typing in the code in this book is the programming equivalent of repeating phrases back to an instructor as you're learning to speak a new language.

Expressions and Values: Arithmetic in Python

You're familiar with mathematical expressions like `3 + 4` ("three plus four") and `2 - 3 / 5` ("two minus three divided by five"); each expression is built out of *values* like `2`, `3`, and `5` and *operators* like `+` and `-`, which combine their *operands* in different ways. In the expression `4 / 5`, the operator is "/" and the operands are `4` and `5`.

Expressions don't have to involve an operator: a number by itself is an expression. For example, we consider `212` to be an expression as well as a value.

Like any programming language, Python can *evaluate* basic mathematical expressions. For example, the following expression adds `4` and `13`:

```
>>> 4 + 13
17
```

The `>>>` symbol is called a *prompt*. When you opened IDLE, a window should have opened with this symbol shown; you don't type it. It is prompting you to type something. Here we typed `4 + 13`, and then we pressed the Return (or Enter) key in order to signal that we were done entering that *expression*. Python then evaluated the expression.

When an expression is evaluated, it produces a single value. In the previous expression, the evaluation of `4 + 13` produced the value `17`. When you type the expression in the shell, Python shows the value that is produced.

Subtraction and multiplication are similarly unsurprising:

```
>>> 15 - 3
12
>>> 4 * 7
28
```

The following expression divides 5 by 2:

```
>>> 5 / 2
2.5
```

The result has a decimal point. In fact, the result of division always has a decimal point even if the result is a whole number:

```
>>> 4 / 2
2.0
```

Types

Every value in Python has a particular *type*, and the types of values determine how they behave when they're combined. Values like 4 and 17 have type int (short for *integer*), and values like 2.5 and 17.0 have type float. The word *float* is short for *floating point*, which refers to the decimal point that moves around between digits of the number.

An expression involving two floats produces a float:

```
>>> 17.0 - 10.0
7.0
```

When an expression's operands are an int and a float, Python automatically converts the int to a float. This is why the following two expressions both return the same answer:

```
>>> 17.0 - 10
7.0
>>> 17 - 10.0
7.0
```

If you want, you can omit the zero after the decimal point when writing a floating-point number:

```
>>> 17 - 10.
7.0
>>> 17. - 10
7.0
```

However, most people think this is bad style, since it makes your programs harder to read: it's very easy to miss a dot on the screen and see 17 instead of 17..

Integer Division, Modulo, and Exponentiation

Every now and then, we want only the integer part of a division result. For example, we might want to know how many 24-hour days there are in 53 hours (which is two 24-hour days plus another 5 hours). To calculate the number of days, we can use *integer division*:

```
>>> 53 // 24
2
```

We can find out how many hours are left over using the *modulo* operator, which gives the remainder of the division:

```
>>> 53 % 24
5
```

Python doesn't round the result of integer division. Instead, it takes the *floor* of the result of the division, which means that it rounds down to the nearest integer:

```
>>> 17 // 10
1
```

Be careful about using % and // with negative operands. Because Python takes the floor of the result of an integer division, the result is one smaller than you might expect if the result is negative:

```
>>> -17 // 10
-2
```

When using modulo, the sign of the result matches the sign of the divisor (the second operand):

```
>>> -17 % 10
3
>>> 17 % -10
-3
```

For the mathematically inclined, the relationship between // and % comes from this equation, for any two non-zero numbers `a` and `b`:

```
(b * (a // b) + a % b) is equal to a
```

For example, because `-17 // 10` is `-2`, and `-17 % 10` is `3`; then `10 * (-17 // 10) + -17 % 10` is the same as `10 * -2 + 3`, which is `-17`.

Floating-point numbers can be operands for // and % as well. With //, division is performed and the result is rounded down to the nearest whole number, although the type is a floating-point number:

```
>>> 3.3 // 1
3.0
>>> 3 // 1.0
3.0
>>> 3 // 1.1
2.0
>>> 3.5 // 1.1
3.0
>>> 3.5 // 1.3
2.0
```

The following expression calculates 3 raised to the 6th power:

```
>>> 3 ** 6
729
```

Operators that have two operands are called *binary operators*. Negation is a *unary operator* because it applies to one operand:

```
>>> -5
-5
>>> --5
5
>>> ---5
-5
```

What *Is* a Type?

We've now seen two types of numbers (integers and floating-point numbers), so we ought to explain what we mean by a *type*. In Python, a type consists of two things:

- A set of values
- A set of operations that can be applied to those values

For example, in type int, the values are ..., -3, -2, -1, 0, 1, 2, 3, ... and we have seen that these operators can be applied to those values: +, -, *, /, //, %, and **.

The values in type float are a subset of the real numbers, and it happens that the same set of operations can be applied to float values. We can see what happens when these are applied to various values in Table 1, *Arithmetic Operators*, on page 13. If an operator can be applied to more than one type of value, it is called an *overloaded operator*.

Finite Precision

Floating-point numbers are not exactly the fractions you learned in grade school. For example, look at Python's version of the fractions $^2/_3$ and $^5/_3$:

Symbol	Operator	Example	Result
-	Negation	-5	-5
+	Addition	11 + 3.1	14.1
-	Subtraction	5 - 19	-14
*	Multiplication	8.5 * 4	34.0
/	Division	11 / 2	5.5
//	Integer Division	11 // 2	5
%	Remainder	8.5 % 3.5	1.5
**	Exponentiation	2 ** 5	32

Table 1—Arithmetic Operators

```
>>> 2 / 3
0.6666666666666666
>>> 5 / 3
1.6666666666666667
```

The first value ends with a 6, and the second with a 7. This is fishy: both of them should have an infinite number of 6s after the decimal point. The problem is that computers have a finite amount of memory, and (to make calculations fast and memory efficient) most programming languages limit how much information can be stored for any single number. The number 0.6666666666666666 turns out to be the closest value to $^2/_3$ that the computer can actually store in that limited amount of memory, and 1.6666666666666667 is as close as we get to the real value of $^5/_3$.

Operator Precedence

Let's put our knowledge of ints and floats to use in converting Fahrenheit to Celsius. To do this, we subtract 32 from the temperature in Fahrenheit and then multiply by $^5/_9$:

```
>>> 212 - 32 * 5 / 9
194.22222222222223
```

Python claims the result is 194.22222222222223 degrees Celsius, when in fact it should be 100. The problem is that multiplication and division have higher *precedence* than subtraction; in other words, when an expression contains a mix of operators, the * and / are evaluated before the - and +. This means that what we actually calculated was 212 - ((32 * 5) / 9): the *subexpression* 32 * 5 is evaluated before the division is applied, and that division is evaluated before the subtraction occurs.

More on Numeric Precision

Integers (values of type int) in Python can be as large or as small as you like. However, float values are only *approximations* to real numbers. For example, $\frac{1}{4}$ can be stored exactly, but as we've already seen, $\frac{2}{3}$ cannot. Using more memory won't solve the problem, though it will make the approximation closer to the real value, just as writing a larger number of 6s after the 0 in 0.666... doesn't make it exactly equal to $\frac{2}{3}$.

The difference between $\frac{2}{3}$ and 0.6666666666666666 may look tiny. But if we use 0.6666666666666666 in a calculation, then the error may get compounded. For example, if we add 1 to $\frac{2}{3}$, the resulting value ends in ...6665, so in many programming languages, $1 + \frac{2}{3}$ is not equal to $\frac{5}{3}$:

```
>>> 2 / 3 + 1
1.6666666666666665
>>> 5 / 3
1.6666666666666667
```

As we do more calculations, the rounding errors can get larger and larger, particularly if we're mixing very large and very small numbers. For example, suppose we add 10000000000 (10 billion) and 0.00000000001 (there are 10 zeros after the decimal point):

```
>>> 10000000000 + 0.00000000001
10000000000.0
```

The result ought to have twenty zeros between the first and last significant digit, but that's too many for the computer to store, so the result is just 10000000000—it's as if the addition never took place. Adding lots of small numbers to a large one can therefore have no effect at all, which is *not* what a bank wants when it totals up the values of its customers' savings accounts.

It's important to be aware of the floating-point issue. There is no magic bullet to solve it, because computers are limited in both memory and speed. *Numerical analysis*, the study of algorithms to approximate continuous mathematics, is one of the largest subfields of computer science and mathematics.

Here's a tip: If you have to add up floating-point numbers, add them from smallest to largest in order to minimize the error.

We can alter the order of precedence by putting parentheses around subexpressions:

```
>>> (212 - 32) * 5 / 9
100.0
```

Table 2, *Arithmetic Operators Listed by Precedence from Highest to Lowest*, on page 15 shows the order of precedence for arithmetic operators.

Operators with higher precedence are applied before those with lower precedence. Here is an example that shows this:

```
>>> -2 ** 4
-16
>>> -(2 ** 4)
-16
>>> (-2) ** 4
16
```

Because exponentiation has higher precedence than negation, the subexpression `2 ** 4` is evaluated before negation is applied.

Precedence	Operator	Operation
Highest	**	Exponentiation
	-	Negation
	*, /, //, %	Multiplication, division, integer division, and remainder
Lowest	+, -	Addition and subtraction

Table 2—Arithmetic Operators Listed by Precedence from Highest to Lowest

Operators on the same row have equal precedence and are applied left to right, except for exponentiation, which is applied right to left. So, for example, because binary operators + and - are on the same row, `3 + 4 - 5` is equivalent to `(3 + 4) - 5`, and `3 - 4 + 5` is equivalent to `(3 - 4) + 5`.

It's a good rule to parenthesize complicated expressions even when you don't need to, since it helps the eye read things like `1 + 1.7 + 3.2 * 4.4 - 16 / 3`. On the other hand, it's a good rule to *not* use parentheses in simple expressions such as `3.1 * 5`.

Variables and Computer Memory: Remembering Values

Like mathematicians, programmers frequently name values so that they can use them later. A name that refers to a value is called a *variable*. In Python, variable names can use letters, digits, and the underscore symbol (but they can't start with a digit). For example, `X`, `species5618`, and `degrees_celsius` are all allowed, but `777` isn't (it would be confused with a number), and neither is `no-way!` (it contains punctuation). Variable names are case sensitive, so `ph` and `pH` are two different names.

You create a new variable by *assigning* it a value:

```
>>> degrees_celsius = 26.0
```

This statement is called an *assignment statement*; we say that `degrees_celsius` is *assigned* the value `26.0`. That makes `degrees_celsius` refer to the value `26.0`. We can

use variables anywhere we can use values. Whenever Python sees a variable in an expression, it substitutes the value to which the variable refers:

```
>>> degrees_celsius = 26.0
>>> degrees_celsius
26.0
>>> 9 / 5 * degrees_celsius + 32
78.80000000000001
>>> degrees_celsius / degrees_celsius
1.0
```

Variables are called *variables* because their values can vary as the program executes. We can assign a new value to a variable:

```
>>> degrees_celsius = 26.0
>>> 9 / 5 * degrees_celsius + 32
78.80000000000001
>>> degrees_celsius = 0.0
>>> 9 / 5 * degrees_celsius + 32
32.0
```

Assigning a value to a variable that already exists doesn't create a second variable. Instead, the existing variable is reused, which means that the variable no longer refers to its old value.

We can create other variables; this example calculates the difference between the boiling point of water and the temperature stored in degrees_celsius:

```
>>> degrees_celsius = 15.5
>>> difference = 100 - degrees_celsius
>>> difference
84.5
```

Warning: = Is Not Equality in Python!

In mathematics, = means "the thing on the left is equal to the thing on the right." In Python, it means something quite different. Assignment is not symmetric: x = 12 assigns the value 12 to variable x, but 12 = x results in an error. Because of this, we never describe the statement x = 12 as "x equals 12." Instead, we read this as "x gets 12" or "x is assigned 12."

Values, Variables, and Computer Memory

We're going to develop a model of computer memory—a *memory model*—that will let us trace what happens when Python executes a Python program. This memory model will help us accurately predict and explain what Python does when it executes code, a skill that is a requirement for becoming a good programmer.

The Online Python Tutor

Philip Guo wrote a web-based memory visualizer that matches our memory model pretty well. Here's the URL: http://pythontutor.com/visualize.html. It can trace both Python 2 and Python 3 code; make sure you select the correct version. The settings that most closely match our memory model are these:

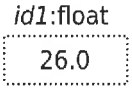
- Hide exited frames
- Render all objects on the heap
- Use text labels for pointers

We strongly recommend that you use this visualizer whenever you want to trace execution of a Python program.

In case you find it motivating, we weren't aware of Philip's visualizer when we developed our memory model (and vice versa), and yet they match extremely closely.

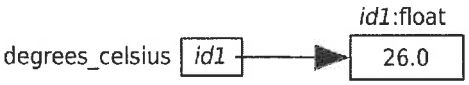
Every location in the computer's memory has a *memory address*, much like an address for a house on a street, that uniquely identifies that location. We're going to mark our memory addresses with an *id* prefix (short for *identifier*) so that they look different from integers: *id1*, *id2*, *id3*, and so on.

Here is how we draw the floating-point value 26.0 using the memory model:



This image shows the value 26.0 at the memory address *id1*. We will always show the type of the value as well—in this case, float. We will call this box an *object*: a value at a memory address with a type. During execution of a program, every value that Python keeps track of is stored inside an object in computer memory.

In our memory model, a variable contains the memory address of the object to which it refers:



In order to make the image easier to interpret, we usually draw arrows from variables to their objects.

We use the following terminology:

- Value 26.0 has the memory address *id1*.
- The object at the memory address *id1* has type float and the value 26.0.

- Variable degrees_celsius *contains* the memory address *id1*.
- Variable degrees_celsius *refers* to the value 26.0.

Whenever Python needs to know which value degrees_celsius refers to, it looks at the object at the memory address that degrees_celsius contains. In this example, that memory address is *id1*, so Python will use the value at the memory address *id1*, which is 26.0.

Assignment Statement

Here is the general form of an assignment statement:

«variable» = «expression»

This is executed as follows:

1. Evaluate the expression on the right of the = sign to produce a value. This value has a memory address.
2. Store the memory address of the value in the variable on the left of the =. Create a new variable if that name doesn't already exist; otherwise, just reuse the existing variable, replacing the memory address that it contains.

Consider this example:

```
>>> degrees_celsius = 26.0 + 5
>>> degrees_celsius
31.0
```

Here is how Python executes the statement degrees_celsius = 26.0 + 5:

1. Evaluate the expression on the right of the = sign: 26.0 + 5. This produces the value 31.0, which has a memory address. (Remember that Python stores all values in computer memory.)
2. Make the variable on the left of the = sign, degrees_celsius, refer to 31.0 by storing the memory address of 31.0 in degrees_celsius.

Reassigning to Variables

Consider this code:

```
>>> difference = 20
>>> double = 2 * difference
>>> double
40
>>> difference = 5
>>> double
40
```

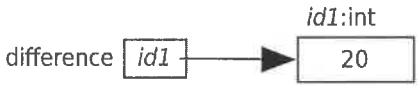
This code demonstrates that assigning to a variable *does not change any other variable*. We start by assigning value 20 to variable difference, and then we assign the result of evaluating 2 * difference (which produces 40) to variable double.

Next, we assign value 5 to variable difference, but when we examine the value of double, it still refers to 40.

Here's how it works according to our rules. The first statement, difference = 20, is executed as follows:

1. Evaluate the expression on the right of the = sign: 20. This produces the value 20, which we'll put at memory address *id1*.
2. Make the variable on the left of the = sign, difference, refer to 20 by storing *id1* in difference.

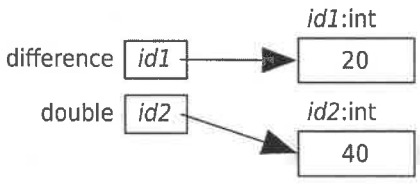
Here is the current state of the memory model. (Variable double has not yet been created because we have not yet executed the assignment to it.)



The second statement, double = 2 * difference, is executed as follows:

1. Evaluate the expression on the right of the = sign: 2 * difference. As we see in the memory model, difference refers to the value 20, so this expression is equivalent to 2 * 20, which produces 40. We'll pick the memory address *id2* for the value 40.
2. Make the variable on the left of the = sign, double, refer to 40 by storing *id2* in double.

Here is the current state of the memory model:

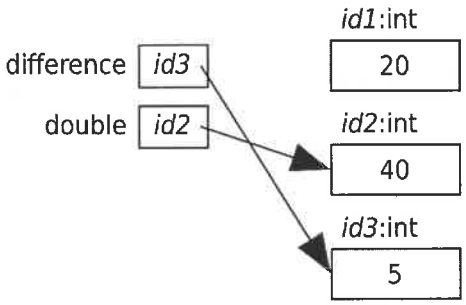


When Python executes the third statement, double, it merely looks up the value that double refers to (40) and displays it.

The fourth statement, difference = 5, is executed as follows:

1. Evaluate the expression on the right of the = sign: 5. This produces the value 5, which we'll put at the memory address *id3*.
2. Make the variable on the left of the = sign, difference, refer to 5 by storing *id3* in difference.

Here is the current state of the memory model:



Variable double still contains *id2*, so it still refers to 40. Neither variable refers to 20 anymore.

The fifth and last statement, double, merely looks up the value that double refers to, which is still 40, and displays it.

We can even use a variable on both sides of an assignment statement:

```
>>> number = 3
>>> number
3
>>> number = 2 * number
>>> number
6
>>> number = number * number
>>> number
36
```

We'll now explain how Python executes this code, but we won't explicitly mention memory addresses. Trace this on a piece of paper while we describe what happens; make up your own memory addresses as you do this.

Python executes the first statement, number = 3, as follows:

1. Evaluate the expression on the right of the = sign: 3. This one is easy to evaluate: 3 is produced.
2. Make the variable on the left of the = sign, number, refer to 3.

Python executes the second statement, number = 2 * number, as follows:

1. Evaluate the expression on the right of the = sign: 2 * number. number currently refers to 3, so this is equivalent to 2 * 3, and 6 is produced.
2. Make the variable on the left of the = sign, number, refer to 6.

Python executes the third statement, number = number * number, as follows:

1. Evaluate the expression on the right of the = sign: number * number. number currently refers to 6, so this is equivalent to 6 * 6, and 36 is produced.
2. Make the variable on the left of the = sign, number, refer to 36.

Augmented Assignment

In this example, the variable score appears on both sides of the assignment statement:

```
>>> score = 50
>>> score
50
>>> score = score + 20
>>> score
70
```

This is so common that Python provides a shorthand notation for this operation:

```
>>> score = 50
>>> score
50
>>> score += 20
>>> score
70
```

An *augmented assignment* combines an assignment statement with an operator to make the statement more concise. An augmented assignment statement is executed as follows:

1. Evaluate the expression on the right of the = sign to produce a value.
2. Apply the operator attached to the = sign to the variable on the left of the = and the value that was produced. This produces another value. Store the memory address of that value in the variable on the left of the =.

Note that the operator is applied *after* the expression on the right is evaluated:

```
>>> d = 2
>>> d *= 3 + 4
>>> d
14
```

All the operators (except for negation) in Table 2, *Arithmetic Operators Listed by Precedence from Highest to Lowest*, on page 15, have shorthand versions. For example, we can square a number by multiplying it by itself:

```
>>> number = 10
>>> number *= number
>>> number
100
```

This code is equivalent to this:

```
>>> number = 10
>>> number = number * number
>>> number
100
```

Table 3 contains a summary of the augmented operators you've seen plus a few more based on arithmetic operators you learned about in *Expressions and Values: Arithmetic in Python*, on page 9.

Symbol	Example	Result
+=	`x = 7` `x += 2`	x refers to 9
-=	`x = 7` `x -= 2`	x refers to 5
*=	`x = 7` `x *= 2`	x refers to 14
/=	`x = 7` `x /= 2`	x refers to 3.5
//=	`x = 7` `x //= 2`	x refers to 3
%=	`x = 7` `x %= 2`	x refers to 1
**=	`x = 7` `x **= 2`	x refers to 49

Table 3—Augmented Assignment Operators

How Python Tells You Something Went Wrong

Broadly speaking, there are two kinds of errors in Python: *syntax errors*, which happen when you type something that isn't valid Python code, and *semantic errors*, which happen when you tell Python to do something that it just can't do, like divide a number by zero or try to use a variable that doesn't exist.

Here is what happens when we try to use a variable that hasn't been created yet:

```
>>> 3 + moogah
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'moogah' is not defined
```

This is pretty cryptic; Python error messages are meant for people who already know Python. (You'll get used to them and soon find them helpful.) The first two lines aren't much use right now, though they'll be indispensable when we start writing longer programs. The last line is the one that tells us what went wrong: the name `moogah` wasn't recognized.

Here's another error message you might sometimes see:

```
>>> 2 +
  File "<stdin>", line 1
    2 +
      ^
SyntaxError: invalid syntax
```

The rules governing what is and isn't legal in a programming language are called its *syntax*. The message tells us that we violated Python's syntax rules—in this case, by asking it to add something to `2` but not telling it what to add.

Earlier, in *Warning: = Is Not Equality in Python!*, on page 16, we claimed that `12 = x` results in an error. Let's try it:

```
>>> 12 = x
  File "<stdin>", line 1
SyntaxError: can't assign to literal
```

A *literal* is any value, like `12` and `26.0`. This is a `SyntaxError` because when Python examines that assignment statement, it knows that you can't assign a value to a number even before it tries to execute it; you can't change the value of `12` to anything else. `12` is just `12`.

A Single Statement That Spans Multiple Lines

Sometimes statements get pretty intricate. The recommended Python style is to limit lines to 80 characters, including spaces, tabs, and other *whitespace* characters, and that's a common limit throughout the programming world. Here's what to do when lines get too long or when you want to split it up for clarity.

In order to split up a statement into more than one line, you need to do one of two things:

1. Make sure your line break occurs inside parentheses.
2. Use the line-continuation character, which is a backslash, \.

Note that the line-continuation character is a backslash (\), not the division symbol (/).

Here are examples of both:

```
>>> (2 +
... 3)
5
>>> 2 + \
... 3
5
```

Notice how we don't get a SyntaxError. Each triple-dot prompt in our examples indicates that we are in the middle of entering an expression; we use them to make the code line up nicely. You do not type the dots any more than you type the greater-than signs in the usual >>> prompt, and if you are using IDLE, you won't see them at all.

Here is a more realistic (and tastier) example: let's say we're baking cookies. The authors live in Canada, which uses Celsius, but we own cookbooks that use Fahrenheit. We are wondering how long it will take to preheat our oven. Here are our facts:

- The room temperature is 20 degrees Celsius.
- Our oven controls use Celsius, and the oven heats up at 20 degrees per minute.
- Our cookbook uses Fahrenheit, and it says to preheat the oven to 350 degrees.

We can convert t degrees Fahrenheit to t degrees Celsius like this: (t - 32) * 5 / 9. Let's use this information to try to solve our problem.

```
>>> room_temperature_c = 20
>>> cooking_temperature_f = 350
>>> oven_heating_rate_c = 20
>>> oven_heating_time = (
... ((cooking_temperature_f - 32) * 5 / 9) - room_temperature_c) / \
... oven_heating_rate_c
>>> oven_heating_time
7.833333333333333
```

Not bad—just under eight minutes to preheat.

The assignment statement to variable oven_heating_time spans three lines. The first line ends with an open parenthesis, so we do not need a line-continuation character. The second ends *outside* the parentheses, so we need the line-continuation character. The third line completes the assignment statement.

That's still hard to read. Once we've continued an expression on the next line, we can indent (by pressing the Tab key or by pressing the spacebar a bunch) to our heart's content to make it clearer:

```
>>> oven_heating_time = (
...     ((cooking_temperature_f - 32) * 5 / 9) - room_temperature_c) / \
...     oven_heating_rate_c
```

Or even this—notice how the two subexpressions involved in the subtraction line up:

```
>>> oven_heating_time = (
...     ((cooking_temperature_f - 32) * 5 / 9) -
...      room_temperature_c) / \
...     oven_heating_rate_c
```

In the previous example, we clarified the expression by working with indentation. However, we could have made this process even clearer by converting the cooking temperature to Celsius before calculating the heating time:

```
>>> room_temperature_c = 20
>>> cooking_temperature_f = 350
>>> cooking_temperature_c = (cooking_temperature_f - 32) * 5 / 9
>>> oven_heating_rate_c = 20
>>> oven_heating_time = (cooking_temperature_c - room_temperature_c) / \
...     oven_heating_rate_c
>>> oven_heating_time
7.833333333333333
```

The message to take away here is that well-named temporary variables can make code much clearer.

Describing Code

Programs can be quite complicated and are often thousands of lines long. It can be helpful to write a *comment* describing parts of the code so that when you or someone else reads it the meaning is clear.

In Python, any time the # character is encountered, Python will ignore the rest of the line. This allows you to write English sentences:

```
>>> # Python ignores this sentence because of the # symbol.
```

The # symbol does not have to be the first character on the line; it can appear at the end of a statement:

```
>>> (212 - 32) * 5 / 9 # Convert 212 degrees Fahrenheit to Celsius.
100.0
```

Notice that the comment doesn't describe how Python works. Instead, it is meant for humans reading the code to help them understand why the code exists.

Making Code Readable

Much like there are spaces in English sentences to make the words easier to read, we use spaces in Python code to make it easier to read. In particular, we always put a space before and after every binary operator. For example, we write `v = 4 + -2.5 / 3.6` instead of `v=4+-2.5/3.6`. There are situations where it may not make a difference, but that's a detail we don't want to fuss about, so we always do it: it's almost never *harder* to read if there are spaces.

Psychologists have discovered that people can keep track of only a handful of things at any one time (*Forty Studies That Changed Psychology [Hoc04]*). Since programs can get quite complicated, it's important that you choose names for your variables that will help you remember what they're for. `id1`, `X2`, and `blah` won't remind you of anything when you come back to look at your program next week: use names like `celsius`, `average`, and `final_result` instead.

Other studies have shown that your brain automatically notices differences between things—in fact, there's no way to stop it from doing this. As a result, the more inconsistencies there are in a piece of text, the longer it takes to read. (JuSt thInK a bout how long It w o u l d tAKE you to rEa d this cHaPTer iF IT wAs fORmaTTeD like thIs.) It's therefore also important to use consistent names for variables. If you call something `maximum` in one place, don't call it `max_val` in another; if you use the name `max_val`, don't also use the name `maxVal`, and so on.

These rules are so important that many programming teams require members to follow a style guide for whatever language they're using, just as newspapers and book publishers specify how to capitalize headings and whether to use a comma before the last item in a list. If you search the Internet for *programming style guide* (https://www.google.com/search?q=programming+style+guide), you'll discover links to hundreds of examples. In this book, we follow the style guide for Python from http://www.python.org/dev/peps/pep-0008/.

You will also discover that lots of people have wasted many hours arguing over what the "best" style for code is. Some of your classmates (and your

instructors) may have strong opinions about this as well. If they do, ask them what data they have to back up their beliefs. Strong opinions need strong evidence to be taken seriously.

The Object of This Chapter

In this chapter, you learned the following:

- An operating system is a program that manages your computer's hardware on behalf of other programs. An interpreter or virtual machine is a program that sits on top of the operating system and runs your programs for you. The Python shell is an interpreter, translating your Python statements into language the operating system understands and translating the results back so you can see and use them.
- Programs are made up of statements, or instructions. These can be simple expressions like `3 + 4` and assignment statements like `celsius = 20` (which create new variables or change the values of existing ones). There are many other kinds of statements in Python, and we'll introduce them throughout the book.
- Every value in Python has a specific type, which determines what operations can be applied to it. The two types used to represent numbers are `int` and `float`. Floating-point numbers are approximations to real numbers.
- Python evaluates an expression by applying higher-precedence operators before lower-precedence operators. You can change that order by putting parentheses around subexpressions.
- Python stores every value in computer memory. A memory location containing a value is called an object.
- Variables are created by executing assignment statements. If a variable already exists because of a previous assignment statement, Python will use that one instead of creating a new one.
- Variables contain memory addresses of values. We say that variables refer to values.
- Variables must be assigned values before they can be used in expressions.

Exercises

Here are some exercises for you to try on your own. Solutions are available at http://pragprog.com/titles/gwpy3/practical-programming.

1. For each of the following expressions, what value will the expression give? Verify your answers by typing the expressions into Python.

   a. `9 - 3`

   b. `8 * 2.5`

   c. `9 / 2`

   d. `9 / -2`

   e. `9 // -2`

   f. `9 % 2`

   g. `9.0 % 2`

   h. `9 % 2.0`

   i. `9 % -2`

   j. `-9 % 2`

   k. `9 / -2.0`

   l. `4 + 3 * 5`

   m. `(4 + 3) * 5`

2. Unary minus negates a number. Unary plus exists as well; for example, Python understands `+5`. If `x` has the value `-17`, what do you think `+x` should do? Should it leave the sign of the number alone? Should it act like absolute value, removing any negation? Use the Python shell to find out its behavior.

3. Write two assignment statements that do the following:

   a. Create a new variable, `temp`, and assign it the value `24`.

   b. Convert the value in `temp` from Celsius to Fahrenheit by multiplying by `1.8` and adding `32`; make `temp` refer to the resulting value.

   What is `temp`'s new value?

4. For each of the following expressions, in which order are the subexpressions evaluated?

   a. `6 * 3 + 7 * 4`

   b. `5 + 3 / 4`

   c. `5 - 2 * 3 ** 4`

5. a. Create a new variable x, and assign it the value 10.5.

   b. Create a new variable y, and assign it the value 4.

   c. Sum x and y, and make x refer to the resulting value. After this statement has been executed, what are the values of x and y?

6. Write a bullet list description of what happens when Python evaluates the statement x += x - x when x has the value 3.

7. When a variable is used before it has been assigned a value, a NameError occurs. In the Python shell, write an expression that results in a NameError.

8. Which of the following expressions results in SyntaxErrors?

   a. 6 * ----------8

   b. 8 = people

   c. ((((4 ** 3))))

   d. (-(-(-(-5))))

   e. 4 += 7 / 2