

# 2

## PROGRAMMING IN THE INTERACTIVE SHELL

*“The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform.”*

—Ada Lovelace, October 1842



Before you can write encryption programs, you need to learn some basic programming concepts. These concepts include values, operators, expressions, and variables.

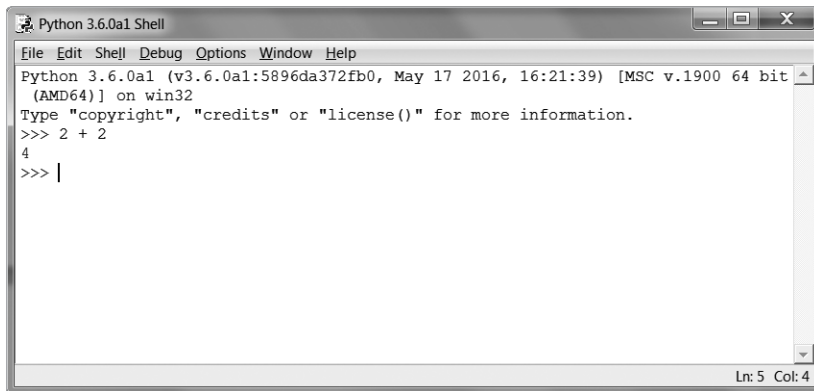
### TOPICS COVERED IN THIS CHAPTER

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

Let's start by exploring how to do some simple math in Python's interactive shell. Be sure to read this book next to your computer so you can enter the short code examples and see what they do. Developing muscle memory from typing programs will help you remember how Python code is constructed.

## Some Simple Math Expressions

Start by opening IDLE (see "Starting IDLE" on page xi). You'll see the interactive shell and the cursor blinking next to the `>>>` prompt. The interactive shell can work just like a calculator. Type `2 + 2` into the shell and press ENTER on your keyboard. (On some keyboards, this is the RETURN key.) The computer should respond by displaying the number 4, as shown in Figure 2-1.



```
Python 3.6.0a1 Shell
File Edit Shell Debug Options Window Help
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 2-1: Type `2 + 2` into the shell.

In the example in Figure 2-1, the `+` sign tells the computer to add the numbers 2 and 2, but Python can do other calculations as well, such as subtract numbers using the minus sign (`-`), multiply numbers with an asterisk (`*`), or divide numbers with a forward slash (`/`). When used in this way, `+`, `-`, `*`, and `/` are called *operators* because they tell the computer to perform an operation on the numbers surrounding them. Table 2-1 summarizes the Python math operators. The 2s (or other numbers) are called *values*.

Table 2-1: Math Operators in Python

Operator	Operation
<code>+</code>	Addition
<code>-</code>	Subtraction
<code>*</code>	Multiplication
<code>/</code>	Division

By itself, `2 + 2` isn't a program; it's just a single instruction. Programs are made of many of these instructions.

## Integers and Floating-Point Values

In programming, whole numbers, such as 4, 0, and 99, are called *integers*. Numbers with decimal points (3.5, 42.1, and 5.0) are called *floating-point numbers*. In Python, the number 5 is an integer, but if you wrote it as 5.0, it would be a floating-point number.

Integers and floating points are *data types*. The value 42 is a value of the integer, or *int*, data type. The value 7.5 is a value of the floating point, or *float*, data type.

Every value has a data type. You'll learn about a few other data types (such as strings in Chapter 3), but for now just remember that any time we talk about a value, that value is of a certain data type. It's usually easy to identify the data type just by looking at how the value is written. Ints are numbers without decimal points. Floats are numbers with decimal points. So 42 is an int, but 42.0 is a float.

## Expressions

You've already seen Python solve one math problem, but Python can do a lot more. Try typing the following math problems into the shell, pressing the ENTER key after each one:

---

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

---

These math problems are called *expressions*. Computers can solve millions of these problems in seconds. Expressions are made up of values (the numbers) connected by operators (the math signs), as shown in Figure 2-2. You can have as many numbers in an expression as you want ❶, as long as they're connected by operators; you can even use multiple types of operators in a single expression ❷. You can also enter any number of spaces between the integers and these operators ❸. But be sure to always start an expression at the beginning of the line, with no spaces in front, because spaces at the beginning of a line change how Python interprets instructions. You'll learn more about spaces at the beginning of a line in "Blocks" on page 45.

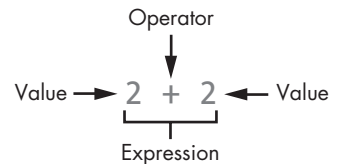


Figure 2-2: An expression is made up of values (like 2) and operators (like +).

## Order of Operations

You might remember the phrase “order of operations” from your math class. For example, multiplication is done before addition. The expression  $2 + 4 * 3$  evaluates to 14 because multiplication is done first to evaluate  $4 * 3$ , and then 2 is added. Parentheses can make different operators go first. In the expression  $(2 + 4) * 3$ , the addition is done first to evaluate  $(2 + 4)$ , and then that sum is multiplied by 3. The parentheses make the expression evaluate to 18 instead of 14. The order of operations (also called *precedence*) of Python math operators is similar to that of mathematics. Operations inside parentheses are evaluated first; next the  $*$  and  $/$  operators are evaluated from left to right; and then the  $+$  and  $-$  operators are evaluated from left to right.

## Evaluating Expressions

When a computer solves the expression  $10 + 5$  and gets the value 15, we say 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.

The expressions  $10 + 5$  and  $10 + 3 + 2$  have the same value, because they both evaluate to 15. Even single values are considered expressions: the expression 15 evaluates to the value 15.

Python continues to evaluate an expression until it becomes a single value, as in the following:

$$\begin{array}{c}
 (5 - 1) * ((7 + 1) / (3 - 1)) \\
 \downarrow \\
 4 * ((7 + 1) / (3 - 1)) \\
 \downarrow \\
 4 * ((8) / (3 - 1)) \\
 \downarrow \\
 4 * ((8) / (2)) \\
 \downarrow \\
 4 * 4.0 \\
 \downarrow \\
 16.0
 \end{array}$$

Python evaluates an expression starting with the innermost, leftmost parentheses. Even when parentheses are nested in each other, the parts of expressions inside them are evaluated with the same rules as any other expression. So when Python encounters  $((7 + 1) / (3 - 1))$ , it first solves the expression in the leftmost inner parentheses,  $(7 + 1)$ , and then solves the expression on the right,  $(3 - 1)$ . When each expression in the inner parentheses is reduced to a single value, the expressions in the outer parentheses are then evaluated. Notice that division evaluates to a floating-point value. Finally, when there are no more expressions in parentheses, Python performs any remaining calculations in the order of operations.

In an expression, you can have two or more values connected by operators, or you can have just one value, but if you enter one value and an operator into the interactive shell, you'll get an error message:

---

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

---

This error happens because `5 +` is not an expression. Expressions with multiple values need operators to connect those values, and in the Python language, the `+` operator expects to connect two values. A *syntax error* means that the computer doesn't understand the instruction you gave it because you typed it incorrectly. This may not seem important, but computer programming isn't just about telling the computer what to do—it's also about knowing the correct way to give the computer instructions that it can follow.

### ERRORS ARE OKAY!

It's perfectly fine to make errors! You won't break your computer by entering code that causes errors. Python will simply tell you an error has occurred and then display the `>>>` prompt again. You can continue entering new code into the interactive shell.

Until you gain more programming experience, error messages might not make a lot of sense to you. However, you can always google the error message text to find web pages that explain that specific error. You can also go to <https://www.nostarch.com/crackingcodes/> to see a list of common Python error messages and their meanings.

## Storing Values with Variables

Programs often need to save values to use later in the program. You can store values in *variables* by using the `=` sign (called the *assignment operator*). For example, to store the value 15 in a variable named `spam`, enter `spam = 15` into the shell:

---

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

---

You can think of the variable like a box with the value 15 inside it (as shown in Figure 2-3). The variable name `spam` is the label on the box (so we can tell one variable from another), and the value stored in it is like a note inside the box.

When you press ENTER, you won't see anything except a blank line in response. Unless you see an error message, you can assume that the instruction executed successfully. The next `>>>` prompt appears so you can enter the next instruction.

This instruction with the = assignment operator (called an *assignment statement*) creates the variable `spam` and stores the value 15 in it. Unlike expressions, *statements* are instructions that don't evaluate to any value; instead, they just perform an action. This is why no value is displayed on the next line in the shell.

Figuring out which instructions are expressions and which are statements might be confusing. Just remember that if a Python instruction evaluates to a single value, it's an expression. If it doesn't, it's a statement.

An assignment statement is written as a variable, followed by the = operator, followed by an expression, as shown in Figure 2-4. The value that the expression evaluates to is stored inside the variable.

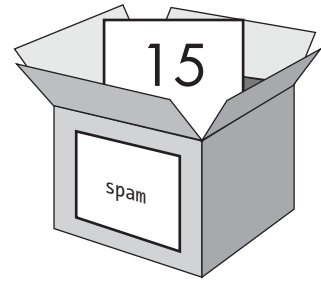


Figure 2-3: Variables are like boxes with names that can hold value.

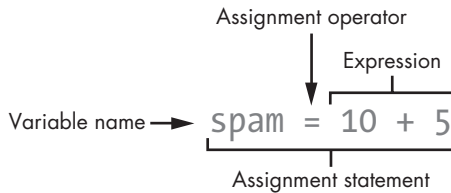


Figure 2-4: The parts of an assignment statement

Keep in mind that variables store single values, not the expressions they are assigned. For example, if you enter the statement `spam = 10 + 5`, the expression `10 + 5` is first evaluated to 15 and then the value 15 is stored in the variable `spam`, as we can see by entering the variable name into the shell:

---

```
>>> spam = 10 + 5
>>> spam
15
```

---

A variable by itself is an expression that evaluates to the value stored in the variable. A value by itself is also an expression that evaluates to itself:

---

```
>>> 15
15
```

---

And here's an interesting twist. If you now enter `spam + 5` into the shell, you'll get the integer 20:

---

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

---

As you can see, variables can be used in expressions the same way values can. Because the value of `spam` is 15, the expression `spam + 5` evaluates to the expression `15 + 5`, which then evaluates to 20.

## Overwriting Variables

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

---

```
>>> spam = 15
❶ >>> spam + 5
❷ 20
❸ >>> spam = 3
❹ >>> spam + 5
❺ 8
```

---

The first time you enter `spam + 5` ❶, the expression evaluates to 20 ❷ because you stored the value 15 inside the variable `spam`. But when you enter `spam = 3` ❸, the value 15 is *overwritten* (that is, replaced) with the value 3, as shown in Figure 2-5. Now when you enter `spam + 5` ❹, the expression evaluates to 8 ❺ because `spam + 5` evaluates to `3 + 5`. The old value in `spam` is forgotten.

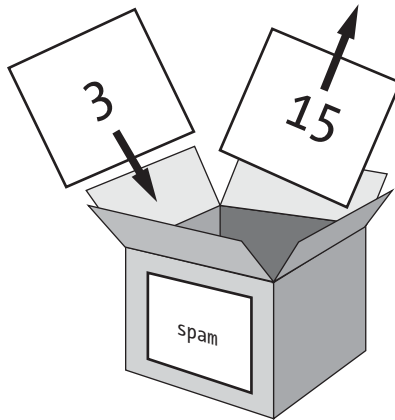


Figure 2-5: The value 15 in `spam` is overwritten by the value 3.

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

---

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

---

The assignment statement `spam = spam + 5` tells the computer that “the new value of the `spam` variable is the current value of `spam` plus five.” The variable on the left side of the `=` sign is assigned the value of the expression on the right side. You can keep increasing the value in `spam` by 5 several times:

---

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

---

The value in `spam` is changed each time `spam = spam + 5` is executed. The value stored in `spam` ends up being 30.

### **Variable Names**

Although the computer doesn’t care what you name your variables, you should. Giving variables names that reflect what type of data they contain makes it easier to understand what a program does. You could give your variables names like `abrahamLincoln` or `monkey` even if your program had nothing to do with Abraham Lincoln or monkeys—the computer would still run the program (as long as you consistently used `abrahamLincoln` or `monkey`). But when you return to a program after not seeing it for a long time, you might not remember what each variable does.

A good variable name describes the data it contains. Imagine that you moved to a new house and labeled all of your moving boxes *Stuff*. You’d never find anything! The variable names `spam`, `eggs`, `bacon`, and so on (inspired by the *Monty Python* “Spam” sketch) are used as generic names for the examples in this book and in much of Python’s documentation, but in your programs, a descriptive name helps make your code more readable.

Variable names (as well as everything else in Python) are case sensitive. *Case sensitive* means the same variable name in a different case is considered an entirely different variable. For example, `spam`, `SPAM`, `Spam`, and `sPAM` are considered four different variables in Python. They each can contain their own separate values and can’t be used interchangeably.

## **Summary**

So when are we going to start making encryption programs? Soon. But before you can hack ciphers, you need to learn just a few more basic programming concepts so there’s one more programming chapter you need to read.

In this chapter, you learned the basics of writing Python instructions in the interactive shell. Python needs you to tell it exactly what to do in a way it expects, because computers only understand very simple instructions. You learned that Python can evaluate expressions (that is, reduce the expression to a single value) and that expressions are values (such as 2 or 5) combined with operators (such as `+` or `-`). You also learned that you can store values inside variables so your program can remember them to use later on.



The interactive shell is a useful tool for learning what Python instructions do because it lets you enter them one at a time and see the results. In Chapter 3, you'll create programs that contain many instructions that are executed in sequence rather than one at a time. We'll discuss some more basic concepts, and you'll write your first program!

### PRACTICE QUESTIONS

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

- Which is the operator for division, / or \ ?
- Which of the following is an integer value, and which is a floating-point value?

---

42  
3.141592

---

- Which of the following lines are *not* expressions?

---

4 x 10 + 2  
3 \* 7 + 1  
2 +  
42  
2 + 2  
spam = 42

---

- If you enter the following lines of code into the interactive shell, what do lines ❶ and ❷ print?

---

spam = 20  
❶ spam + 20  
SPAM = 30  
❷ spam

---

