The Pragmatic Programmers

Practical Programming

Third Edition

An Introduction to Computer Science Using Python 3.6

Paul Gries
Jennifer Campbell
Jason Montojo
edited by Tammy Coron

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Programs are composed of commands that instruct the computer on 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?

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

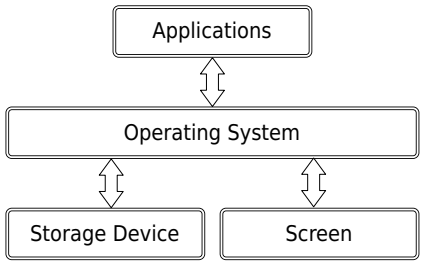
A modern computer is built from several hardware components, including a *processor* (Central Processing Unit, a CPU) that runs programs and performs calculations, storage such as a *solid-state drive* (SSD) to hold data, and other essential parts, such as a touchscreen or monitor, keyboard, and Wi-Fi or cellular connectivity for getting online.

Why Is a CPU "Central"?

Older computers from the 1960s had more than one processing unit, including a central processing unit and peripheral processing units, also known as input/output processors. Modern computers also have more than one processing unit, the others usually being a graphics processing unit (GPU) or a digital signal processor (DSP).

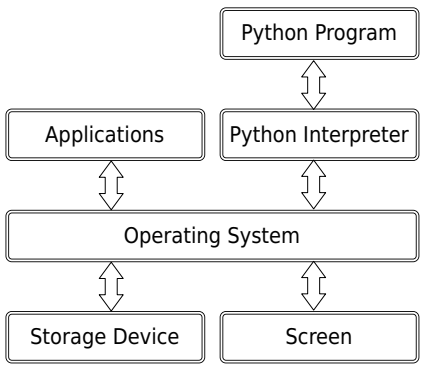
To manage all these components, every computer runs an *operating system* (OS), such as Microsoft Windows, Linux, or macOS. An operating system is one of the few programs that has direct access to the hardware. When any other application (such as your browser, a spreadsheet program, or a game) wants 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:

An operating system is a program that manages your computer's hardware



This arrangement may seem like a convoluted approach to getting things done, such as displaying images on the screen or responding to your actions. Yet, it means that only the people writing the OS have to worry about the differences between one graphics card and another, as well as whether the computer is connected to a network via Ethernet or wireless. The rest of us, analyzing scientific data, creating mobile apps, editing videos, or exploring virtual worlds, 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 intermediary (a layer) between the programmer and the computer's hardware. When you write a program in Python, Java, or Visual Basic, the program doesn't run directly in contact with the OS. Instead, another program called an *interpreter* or *virtual machine* translates your program's commands into a language the OS understands. Command interpretation is easier, more secure, and more portable across operating systems than direct execution, although the latter is somewhat faster:



There are two ways to use the Python interpreter. One is to use a command to execute a Python program saved in a file with a `.py` extension. The other is to interact with the interpreter 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. You will explore Python in this chapter using a Python shell.

Programs are made up of statements

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*

Install Python Now (If You Haven't Already)

If you haven't yet installed Python, please do so now. (Python 2 won't do; there are significant differences between Python 2 and Python 3, and this book uses Python 3.14.)

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

Python comes with a program called IDLE, which you 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 the 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.

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, `212` is 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 start IDLE, the window should open with this symbol displayed; you don't type it. It is prompting you to type something. Type `4 + 13`, and then press the Return (or Enter) key to signal that you are 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, even if it is a whole number:

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

Types

Every value in Python has a particular *type* (a *data 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 a number.

Every value in Python has a specific type

An expression involving two `float`s 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 conversion 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, this omission is considered 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.` (with a period).

Integer Division, Modulo, and Exponentiation

Now and then, you want only the integer part of a division result. For example, you 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, you can use *integer division*:

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

You can determine the number of hours remaining by using the *modulo* operator, which returns 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 (truncates the fractional part):

```
>>> 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 also be operands for // and % operators. 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 power of 6:

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

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

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