Extracted from:

Practical Programming, 2nd Edition

An Introduction to Computer Science Using Python 3

The Pragmatic Bookshelf

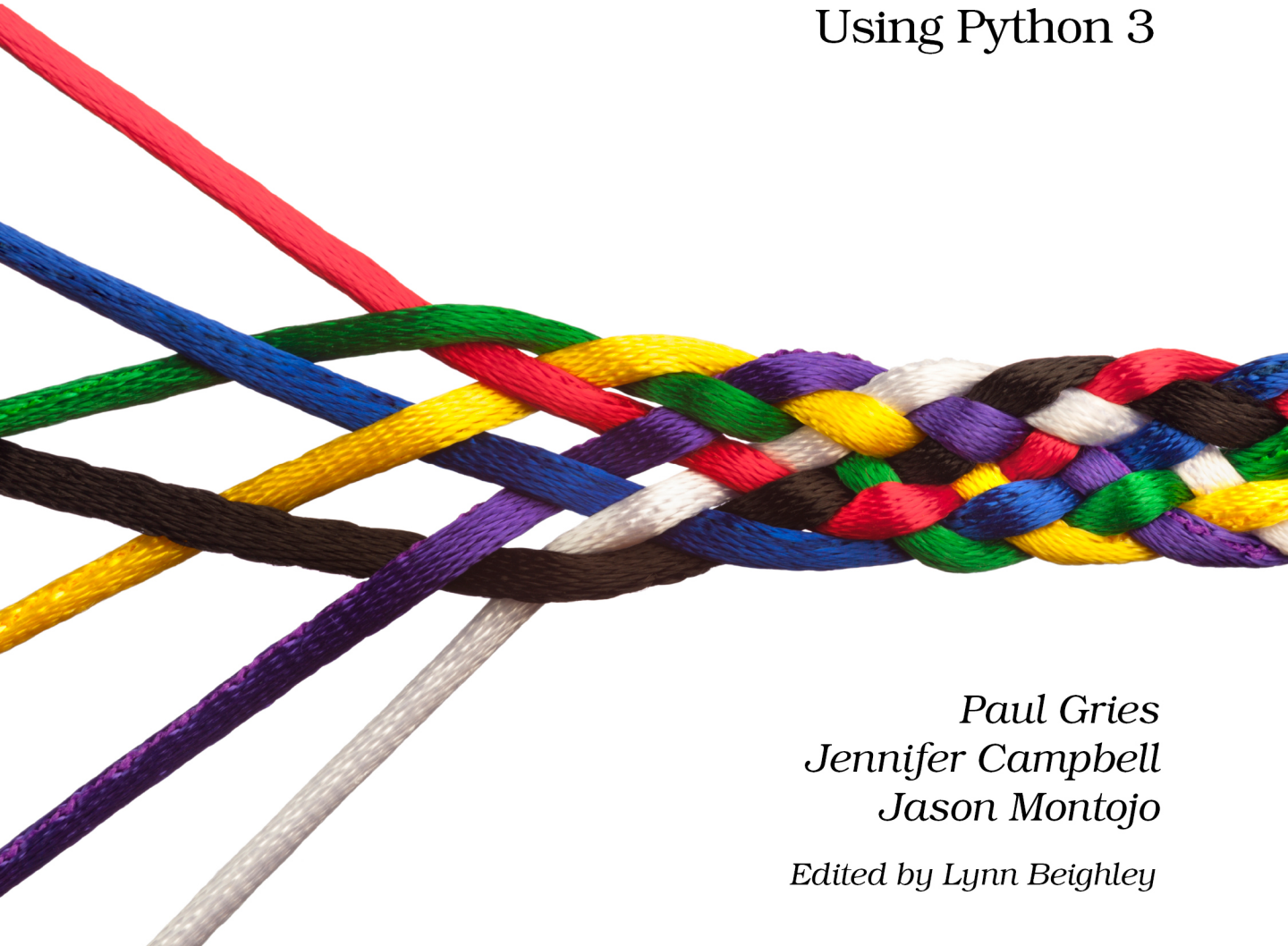
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Practical Programming

Second Edition

An Introduction to Computer Science Using Python 3



Paul Gries
Jennifer Campbell
Jason Montojo

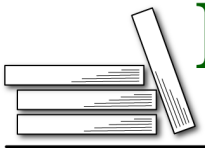
Edited by Lynn Beighley

Practical Programming, 2nd Edition

An Introduction to Computer Science Using Python 3

Paul Gries
Jennifer Campbell
Jason Montojo

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Lynn Beighley (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-93778-545-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2013

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.

2.1 How Does a Computer Run a Python Program?

In order to understand what happens when you're programming, you need to have a basic understanding 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 computer monitor, a keyboard, a card 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 Mac OS X. 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 the hard drive, it sends a request to the OS (see Figure 1, *Talking to the operating system*, on page 6).

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.

Twenty-five years ago that's how most programmers worked. Today, though, 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:

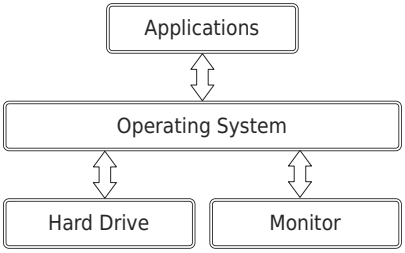
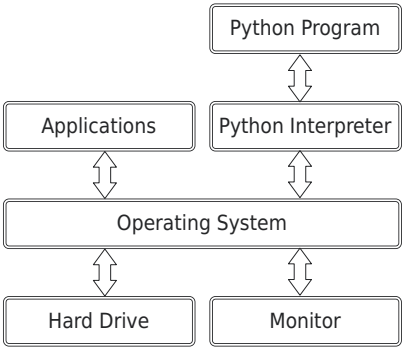


Figure 1—Talking to the operating system



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, 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.) Locate installation instructions on the book's website: http://pragprog.com/titles/gwpy2/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.

2.2 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 typed 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 `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 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 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 //, 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 `6`th 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
```