Beginning\_Python\_Magnus\_01

Beginning Python

From Novice to Professional

Third Edition

Magnus Lie Hetland

pp27

Preface

To quote the old Monty Python song: “Here comes another one / Here it comes again / Here comes another one / When will it ever end?” Since the previous edition, Python 3 has become much more widespread, so this edition has fully transitioned to the Py3 world. There have been other changes as well, with packages in the Python ecosystem coming and going and coding practices going in and out of fashion.

Para citar la vieja canción de Monty Python: "Aquí viene otra / Aquí viene otra vez / Aquí viene otra / ¿Cuándo terminará?" Desde la edición anterior, Python 3 se ha generalizado mucho más, por lo que esta edición ha hecho una transición completa al mundo **Py3**. También ha habido otros cambios, con paquetes en el **ecosistema** de Python que van y vienen y prácticas de codificación que se ponen y pasan de moda.

Where it has been necessary or useful, the book has been rewritten, but its origins are still visible. For example, when the original Practical Python came out at the beginning of the millennium, Usenet was still in semi-widespread use, though nowadays most Internet users probably haven’t even heard of it. So when the fourth code project (Chapter 23) involves connecting to an NNTP server, this is more of a historical curiosity than skills you are likely to apply directly in a mainstream programming career.

Donde ha sido necesario o útil, el libro ha sido reescrito, pero sus orígenes aún son visibles. Por ejemplo, cuando apareció el **Practical Python** original a principios del milenio, Usenet todavía estaba en uso semi generalizado, aunque hoy en día la mayoría de los usuarios de Internet probablemente ni siquiera hayan oído hablar de él. Entonces, cuando el cuarto proyecto de código (Capítulo 23) implica conectarse a un servidor NNTP, esto es más una curiosidad histórica que las habilidades que probablemente aplicará directamente en una carrera de programación convencional.

Still, I’ve kept some of these more quirky parts of the book, as they still work well as programming examples and as they are part of the history of the book. All the people who helped previous editions see the light of day still deserve as many thanks as before.

Aún así, conservé algunas de estas partes más peculiares del libro, ya que todavía funcionan bien como ejemplos de programación y son parte de la historia del libro. Todas las personas que ayudaron a que las ediciones anteriores vieran la luz aún merecen tanto agradecimiento como antes.

This time around, I would in particular like to extend my thanks to Mark Powers, who has been a paragon of patience when my progress faltered. I’d also like to thank Michael Thomas, who has done a great job of checking the technical aspects of the book (... and pointing out all the Python 2–style print statements I had missed; I hope I got them all). I hope you enjoy this updated edition, even though, as Terry Jones says about the song mentioned initially, “Obviously it would be better with a full orchestra.”

Esta vez, en particular, me gustaría extender mi agradecimiento a Mark Powers, quien ha sido un modelo de paciencia cuando mi progreso flaqueó. También me gustaría agradecer a Michael Thomas, quien ha hecho un gran trabajo revisando los aspectos técnicos del libro (... y señalando todas las declaraciones impresas estilo Python 2 que me había perdido; espero haberlas entendido todas) . Espero que disfruten de esta edición actualizada, aunque, como dice Terry Jones sobre la canción mencionada inicialmente, “Obviamente sería mejor con una orquesta completa”.

Preface to the Second Edition

Here it is—a shiny new edition of Beginning Python. If you count its predecessor, Practical Python, this is actually the third edition, and a book I’ve been involved with for the better part of a decade. During this time, Python has seen many interesting changes, and I’ve done my best to update my introduction to the language. At the moment, Python is facing perhaps its most marked transition in a very long time: the introduction of version 3. As I write this, the final release isn’t out yet, but the features are clearly defined and working versions are available. One interesting challenge linked to this language revision is that it isn’t backward-compatible. In other words, it doesn’t simply add features that I could pick and choose from in my writing. It also changes the existing language, so that certain things that are true for Python 2.5 no longer hold.

Had it been clear that the entire Python community would instantly switch to the new version and update all its legacy code, this would hardly be a problem. Simply describe the new language! However, a lot of code written for older versions exists, and much will probably still be written, until version 3 is universally accepted as The Way To Go ™.

So, how have I gotten myself out of this pickle? First of all, even though there are incompatible changes, most of the language remains the same. Therefore, if I wrote entirely about Python 2.5, it would be mostly correct for Python 3 (and even more so for its companion release, 2.6). As for the parts that will no longer be correct, I have been a bit conservative and assumed that full adoption of version 3 will take some time. I have based the book primarily on 2.5 and noted things that will change throughout the text. In addition, I’ve included Appendix D, which gives you an overview of the main changes. I think this will work out for most readers.

In writing this second edition, I have had a lot of help from several people. Just as with the previous two versions (the first edition, and, before it, Practical Python), Jason Gilmore got me started and played an important role in getting the project on the road. As it has moved along, Richard Dal Porto, Frank Pohlmann, and Dominic Shakeshaft have been instrumental in keeping it going. Richard Taylor has certainly played a crucial role in ensuring that the code is correct (and if it still isn’t, I’m the one to blame), and Marilyn Smith has done a great job tuning my writing. My thanks also go out to other Apress staff, including Liz Berry, Beth Christmas, Steve Anglin, and Tina Nielsen, as well as various readers who have provided errata and helpful suggestions, including Bob Helmbold and Waclaw Kusnierczyk. I am also, of course, still thankful to all those who helped in getting the first two incarnations of this book on the shelves.

Introduction

A C program is like a fast dance on a newly waxed dance floor by people carrying razors.

Un programa C es como un baile rápido en una pista de baile recién encerada por personas que llevan navajas de afeitar.

—Waldi Ravens

C++: Hard to learn and built to stay that way.

Difícil de aprender y construida para permanecer así.

(femenino)

—Anonymous

Java is, in many ways, C++ – –.

—Michael Feldman

And now for something completely different . . .

—Monty Python’s Flying Circus

I’ve started this introduction with a few quotes to set the tone for the book, which is rather informal. In

the hope of making it an easy read, I’ve tried to approach the topic of Python programming with a healthy dose of humor, and true to the traditions of the Python community, much of this humor is related to Monty Python sketches.

He comenzado esta introducción con algunas citas para establecer el tono del libro, que es bastante informal. Con la esperanza de que sea una lectura fácil, traté de abordar el tema de la programación de Python con una buena dosis de humor y, fiel a las tradiciones de la comunidad de Python, gran parte de este humor está relacionado con los bocetos de Monty Python.

As a consequence, some of my examples may seem a bit silly; I hope you will bear with me. (And, yes, the name Python is derived from Monty Python, not from snakes belonging to the family Pythonidae.) In this introduction, I give you a quick look at what Python is, why you should use it, who uses it, who this book’s intended audience is, and how the book is organized.

Como consecuencia, algunos de mis ejemplos pueden parecer un poco tontos; Espero que tengas paciencia conmigo. (Y, sí, el nombre Python se deriva de Monty Python, no de las serpientes que pertenecen a la familia Pythonidae). En esta introducción, le doy un vistazo rápido a lo que es Python, por qué debería usarlo, quién lo usa, quién. quién es el público objetivo de este libro y cómo está organizado.

So, what is Python, and why should you use it? To quote an old official blurb, it is “an interpreted, object-oriented, high-level programming language with dynamic semantics.” Many of these terms will become clear as you read this book, but the gist is that Python is a programming language that knows how to stay out of your way when you write your programs. It enables you to implement the functionality you want without any hassle and lets you write programs that are clear and readable (much more so than programs in most other currently popular programming languages).

Entonces, ¿qué es Python y por qué debería usarlo? Para citar una antigua propaganda oficial, es "un lenguaje de programación interpretado, orientado a objetos y de alto nivel con semántica dinámica". Muchos de estos términos se aclararán a medida que lea este libro, pero la esencia es que Python es un lenguaje de programación que sabe cómo permanecer fuera de su camino cuando escribe sus programas. Le permite implementar la funcionalidad que desea sin problemas y le permite escribir programas que son claros y legibles (mucho más que los programas en la mayoría de los otros lenguajes de programación actualmente populares).

Even though Python might not be as fast as compiled languages such as C or C++, what you save in programming time will probably be worth using it, and in most programs, the speed difference won’t be noticeable anyway. If you are a C programmer, you can easily implement the critical parts of your program in C at a later date and have them interoperate with the Python parts. If you haven’t done any programming before (and perhaps are a bit confused by my references to C and C++), Python’s combination of simplicity and power makes it an ideal choice as a place to start.

Aunque Python no sea tan rápido como los lenguajes compilados como C o C++, lo que ahorre en tiempo de programación probablemente valdrá la pena usarlo, y en la mayoría de los programas, la diferencia de velocidad no se notará de todos modos. Si es un programador de C, puede implementar fácilmente las partes críticas de su programa en C en una fecha posterior y hacer que interactúen con las partes de Python. Si no ha hecho nada de programación antes (y tal vez esté un poco confundido por mis referencias a C y C++), la combinación de simplicidad y potencia de Python lo convierte en una opción ideal como punto de partida.

So, who uses Python? Since Guido van Rossum created the language in the early 1990s, its following

has grown steadily, and interest has increased markedly in the past few years. Python is used extensively for system administration tasks (it is, for example, a vital component of several Linux distributions), but it is also used to teach programming to complete beginners.

Entonces, ¿quién usa Python? Desde que Guido van Rossum creó el idioma a principios de la década de 1990, sus seguidores han crecido constantemente y el interés ha aumentado notablemente en los últimos años. Python se usa ampliamente para tareas de administración del sistema (es, por ejemplo, un componente vital de varias distribuciones de Linux), pero también se usa para enseñar programación a principiantes.

The US National Aeronautics and Space Administration (NASA) uses Python both for development and as a scripting language in several of its systems. Industrial Light & Magic uses Python in its production of special effects for large-budget feature films. Yahoo! uses it (among other things) to manage its discussion groups. Google has used it to implement many components of its web crawler and search engine. Python is being used in such diverse areas as computer games and bioinformatics. Soon one might as well ask, “Who isn’t using Python?”

La Administración Nacional de Aeronáutica y del Espacio (NASA) de EE. UU. utiliza Python tanto para el desarrollo como como lenguaje de programación en varios de sus sistemas. Industrial Light & Magic utiliza Python en su producción de efectos especiales para largometrajes de gran presupuesto. yahoo! lo usa (entre otras cosas) para administrar sus grupos de discusión. Google lo ha utilizado para implementar muchos componentes de su rastreador web y motor de búsqueda. Python se está utilizando en áreas tan diversas como los juegos de computadora y la bioinformática. Pronto uno también podría preguntarse: "¿Quién no está usando Python?"

This book is for those of you who want to learn how to program in Python. It is intended to suit a wide

audience, from neophyte programmer to advanced computer wiz. If you have never programmed before, you should start by reading Chapter 1 and continue until you find that things get too advanced for you (if, indeed, they do). Then you should start practicing and write some programs of your own. When the time is right, you can return to the book and proceed with the more intricate stuff.

Este libro es para aquellos de ustedes que quieren aprender a programar en Python. Está destinado a adaptarse a una amplia audiencia, desde programadores neófitos hasta magos informáticos avanzados. Si nunca ha programado antes, debe comenzar leyendo el Capítulo 1 y continuar hasta que descubra que las cosas son demasiado avanzadas para usted (si es que lo son). Entonces deberías empezar a practicar y escribir algunos programas por tu cuenta. Cuando sea el momento adecuado, puede volver al libro y continuar con las cosas más intrincadas.

If you already know how to program, some of the introductory material might not be new to you

(although there will probably be some surprising details here and there). You could skim through the early chapters to get an idea of how Python works, or perhaps read through Appendix A, which is based on my online Python tutorial “Instant Python.” It will get you up to speed on the most important Python concepts.

Si ya sabe cómo programar, es posible que parte del material introductorio no sea nuevo para usted (aunque probablemente habrá algunos detalles sorprendentes aquí y allá). Puede hojear los primeros capítulos para tener una idea de cómo funciona Python, o tal vez leer el Apéndice A, que se basa en mi tutorial en línea de Python "Instant Python". Te pondrá al día con los conceptos más importantes de Python.

After getting the big picture, you could jump straight to Chapter 10 (which describes the Python standard libraries). The last ten chapters present ten programming projects, which show off various capabilities of the Python language. These projects should be of interest to beginners and experts alike. Although some of the material in the later projects may be a bit difficult for an inexperienced programmer, following the projects in order (after reading the material in the first part of the book) should be possible. The projects touch upon a wide range of topics, most of which will be very useful to you when writing programs of your own. You will learn how to do things that may seem completely out of reach to you at this point, such as creating a chat server, a peer-to-peer file-sharing system, or a full-fledged graphical computer game. Although much of the material may seem hard at first glance, I think you will be surprised by how easy most of it really is. If you would like to download the source code, it’s available from the Source Code/Download section of the Apress web site (<http://www.apress.com>). Well, that’s it. I always find long introductions a bit boring myself, so I’ll let you continue with your Pythoneering, either in Chapter 1 or in Appendix A. Good luck, and happy hacking.

Después de ver el panorama general, puede pasar directamente al Capítulo 10 (que describe las **bibliotecas estándar de Python**). Los últimos diez capítulos presentan diez proyectos de programación, que muestran varias capacidades del lenguaje Python. Estos proyectos deberían ser de interés tanto para principiantes como para expertos. Aunque parte del material de los últimos proyectos puede resultar un poco difícil para un programador sin experiencia, debería ser posible seguir los proyectos en orden (después de leer el material de la primera parte del libro). Los proyectos tocan una amplia gama de temas, la mayoría de los cuales le serán muy útiles cuando escriba sus propios programas. Aprenderá cómo hacer cosas que pueden parecer completamente fuera de su alcance en este punto, **como crear un servidor de chat**, un sistema de intercambio de archivos entre pares o un juego de computadora gráfico completo. Aunque gran parte del material puede parecer difícil a primera vista, creo que se sorprenderá de lo fácil que es en realidad la mayor parte. Si desea descargar el código fuente, está disponible en la sección Código fuente/Descargar del sitio web de Apress (http://www.apress.com). Bueno, eso es todo. Siempre encuentro que las introducciones largas son un poco aburridas, así que te dejaré continuar con tu Pitonería, ya sea en el Capítulo 1 o en el Apéndice A. Buena suerte y feliz piratería.

CHAPTER 1

Instant Hacking: The Basics

It’s time to start hacking.1 In this chapter, you learn how to take control of your computer by speaking a language it understands: Python. Nothing here is particularly difficult, so if you know the basic principles of how your computer works, you should be able to follow the examples and try them out yourself. I’ll go through the basics, starting with the excruciatingly simple, but because Python is such a powerful language, you’ll soon be able to do pretty advanced things. To begin, you need to install Python, or verify that you already have it installed. If you’re running macOS or Linux/UNIX, open a terminal (the Terminal app on a Mac), type in python, and press Enter. You should get a welcome message, ending with the following prompt:

Es hora de comenzar a piratear.1 En este capítulo, aprenderá cómo tomar el control de su computadora hablando un lenguaje que entiende: Python. Nada aquí es particularmente difícil, por lo que si conoce los principios básicos de cómo funciona su computadora, debería poder seguir los ejemplos y probarlos usted mismo. Voy a repasar lo básico, comenzando con lo terriblemente simple, pero debido a que Python es un lenguaje tan poderoso, pronto podrá hacer cosas bastante avanzadas. Para comenzar, debe instalar Python o verificar que ya lo tiene instalado. Si está ejecutando macOS o Linux/UNIX, abra una terminal (la aplicación Terminal en una Mac), escriba python y presione Entrar. Debería recibir un mensaje de bienvenida, que termina con el siguiente mensaje:

**>>>**

If you do, you can start entering Python commands immediately. Note, however, that you may have an old version of Python. If the first line starts with Python 2 rather than Python 3, you might want to install a newer version anyway, as Python 3 introduces several breaking changes. The details of the installation process will of course vary with your OS and preferred installation mechanism, but the most straightforward approach is to visit www.python.org, where you should find a link to a download page. It is all pretty self-explanatory—just follow the link to the most recent version for your

platform, be it Windows, macOS, Linux/UNIX, or something else. For Windows and Mac, you’ll download an installer that you can run to actually install Python. For Linux/UNIX, there are source code tarballs that you’ll need to compile yourself, by following the included instructions. If you’re using a package manager such as Homebrew or APT, you can use that to streamline the process.

Once you have Python installed, try to fire up the interactive interpreter. If you’re using the command

line, you could simply use the python command, or perhaps python3 if you have an older version installed as well. If you’d rather use a graphical interface, you can start the IDLE app that comes with the Python installation.

Si lo hace, puede comenzar a ingresar comandos de Python de inmediato. Tenga en cuenta, sin embargo, que es posible que tenga una versión anterior de Python. Si la primera línea comienza con Python 2 en lugar de Python 3, es posible que desee instalar una versión más nueva de todos modos, ya que Python 3 presenta varios cambios importantes. Por supuesto, los detalles del proceso de instalación variarán según su sistema operativo y el mecanismo de instalación preferido, pero el enfoque más sencillo es visitar www.python.org, donde encontrará un enlace a una página de descarga. Todo se explica por sí mismo: simplemente siga el enlace a la versión más reciente para su plataforma, ya sea Windows, macOS, Linux/UNIX u otra. Para Windows y Mac, descargará un instalador que puede ejecutar para instalar Python. Para Linux/UNIX, hay archivos comprimidos de código fuente que deberá compilar usted mismo, siguiendo las instrucciones incluidas. Si está utilizando un administrador de paquetes como Homebrew o APT, puede usarlo para optimizar el proceso. Una vez que haya instalado Python, intente iniciar el intérprete interactivo. Si está utilizando la línea de comando, simplemente puede usar el comando python, o quizás python3 si también tiene instalada una versión anterior. Si prefiere usar una interfaz gráfica, puede iniciar la aplicación IDLE que viene con la instalación de Python.

**The Interactive Interpreter**

When you start up Python, you get a prompt similar to the following:

**Python 3.5.0 (default, Dec 5 2015, 15:03:35)**

**[GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.1.76)] on darwin**

**Type "help", "copyright", "credits" or "license" for more information.**

**>>>**

1Hacking is not the same as cracking, which is a term describing computer crime. The two are often confused, and the usage is gradually changing. Hacking, as I’m using it here, basically means “having fun while programming.

Hackear no es lo mismo que crackear, que es un término que describe los delitos informáticos. Los dos a menudo se confunden y el uso está cambiando gradualmente. Hackear, como lo estoy usando aquí, básicamente significa “divertirse mientras se programa.

The exact appearance of the interpreter and its error messages will depend on which version you are using. This might not seem very interesting, but believe me, it is. This is your gateway to hackerdom—your first step in taking control of your computer. In more pragmatic terms, it’s an interactive Python interpreter. Just to see if it’s working, try the following:

La apariencia exacta del intérprete y sus mensajes de error dependerán de la versión que esté utilizando. Esto puede no parecer muy interesante, pero créanme, lo es. Esta es su puerta de entrada a la piratería: su primer paso para tomar el control de su computadora. En términos más pragmáticos, es un intérprete de Python interactivo. Solo para ver si está funcionando, intente lo siguiente:

>>> print("Hello, world!")

When you press the Enter key, the following output appears:

Hello, world!

>>>

If you are familiar with other computer languages, you may be used to terminating every line with a semicolon. There is no need to do so in Python. A line is a line, more or less. You may add a semicolon if you like, but it won’t have any effect (unless more code follows on the same line), and it is not a common thing to do.

Si está familiarizado con otros lenguajes informáticos, es posible que esté acostumbrado a terminar cada línea con un punto y coma. No hay necesidad de hacerlo en Python. Una línea es una línea, más o menos. Puede agregar un punto y coma si lo desea, pero no tendrá ningún efecto (a menos que siga más código en la misma línea), y no es algo común.

So what happened here? The >>> thingy is the prompt. You can write something in this space, like print "Hello, world!". If you press Enter, the Python interpreter prints out the string “Hello, world!” and you get a new prompt below that. What if you write something completely different? Try it out:

Entonces, ¿qué pasó aquí? La cosa >>> es el indicador. Puede escribir algo en este espacio, como imprimir "¡Hola, mundo!". Si presiona Enter, el intérprete de Python imprime la cadena "¡Hola, mundo!" y obtienes un nuevo mensaje debajo de eso. ¿Qué pasa si escribes algo completamente diferente? Pruébalo:

>>> The Spanish Inquisition

SyntaxError: invalid syntax

>>>

Obviously, the interpreter didn’t understand that.2 (If you are running an interpreter other than IDLE, such as the command-line version for Linux, the error message will be slightly different.) The interpreter also indicates what’s wrong: it will emphasize the word Spanish by giving it a red background (or, in the command-line version, by using a caret, ^).

Obviamente, el intérprete no entendió eso.2 (Si está ejecutando un intérprete que no sea IDLE, como la versión de línea de comandos para Linux, el mensaje de error será ligeramente diferente). El intérprete también indica qué es lo que está mal: enfatizar la palabra español dándole un fondo rojo (o, en la versión de línea de comandos, usando un signo de intercalación, ^).

If you feel like it, play around with the interpreter some more. For some guidance, try entering the command help() at the prompt and pressing Enter. You can press F1 for help about IDLE. Otherwise, let’s press on. After all, the interpreter isn’t much fun when you don’t know what to tell it.

Si te apetece, juega un poco más con el intérprete. Para obtener orientación, intente ingresar el comando help() en el indicador y presione Enter. Puede presionar F1 para obtener ayuda sobre IDLE. De lo contrario, sigamos adelante. Después de todo, el intérprete no es muy divertido cuando no sabes qué decirle.

Algo . . . What?

Before we start programming in earnest, I’ll try to give you an idea of what computer programming is. Simply put, it’s telling a computer what to do. Computers can do a lot of things, but they aren’t very good at thinking for themselves. They really need to be spoon-fed the details. You need to feed the computer an algorithm in some language it understands. Algorithm is just a fancy word for a procedure or recipe—a detailed description of how to do something. Consider the following:

Antes de que comencemos a programar en serio, intentaré darle una idea de lo que es la programación de computadoras. En pocas palabras, le está diciendo a una computadora qué hacer. Las computadoras pueden hacer muchas cosas, pero no son muy buenas para pensar por sí mismas. Realmente necesitan ser alimentados con cuchara los detalles. Necesita alimentar a la computadora con un algoritmo en algún idioma que entienda. Algoritmo es solo una palabra elegante para un procedimiento o receta: una descripción detallada de cómo hacer algo. Considera lo siguiente:

SPAM with SPAM, SPAM, Eggs, and SPAM: First, take some SPAM.

Then add some SPAM, SPAM, and eggs.

If a particularly spicy SPAM is desired, add some SPAM.

Cook until done -- Check every 10 minutes.

Not the fanciest of recipes, but its structure can be quite illuminating. It consists of a series of instructions to be followed in order. Some of the instructions may be done directly (“take some SPAM”), while some require some deliberation (“If a particularly spicy SPAM is desired”), and others must be repeated several times (“Check every 10 minutes.”)

No es la receta más elegante, pero su estructura puede ser muy esclarecedora. Consiste en una serie de instrucciones a seguir en orden. Algunas de las instrucciones se pueden hacer directamente ("toma un poco de SPAM"), mientras que otras requieren cierta deliberación ("Si se desea un SPAM particularmente picante"), y otras deben repetirse varias veces ("Revisar cada 10 minutos").

Recipes and algorithms consist of ingredients (objects, things) and instructions (statements). In this example, SPAM and eggs are the ingredients, while the instructions consist of adding SPAM, cooking for a given length of time, and so on. Let’s start with some reasonably simple Python ingredients and see what you can do with them.

Las recetas y los algoritmos constan de ingredientes (objetos, cosas) e instrucciones (declaraciones). En este ejemplo, los ingredientes son SPAM y huevos, mientras que las instrucciones consisten en agregar SPAM, cocinar durante un tiempo determinado, etc. Comencemos con algunos ingredientes de Python razonablemente simples y veamos qué puede hacer con ellos.

Numbers and Expressions

The interactive Python interpreter can be used as a powerful calculator. Try the following:

El intérprete interactivo de Python se puede utilizar como una potente calculadora. Prueba lo siguiente:

>>> 2 + 2

This should give you the answer 4. That wasn’t too hard. Well, what about this:

Esto debería darte la respuesta 4. Eso no fue demasiado difícil. Bueno, que tal esto:

>>> 53672 + 235253

288925

Still not impressed? Admittedly, this is pretty standard stuff. (I’ll assume that you’ve used a calculator enough to know the difference between 1 + 2 \* 3 and (1 + 2) \* 3.) All the usual arithmetic operators work as expected. Division produces decimal numbers, called floats (or floating-point numbers).

¿Todavía no estás impresionado? Es cierto que esto es algo bastante estándar. (Asumiré que ha usado una calculadora lo suficiente como para saber la diferencia entre 1 + 2 \* 3 y (1 + 2) \* 3). Todos los operadores aritméticos habituales funcionan como se esperaba. La división produce números decimales, llamados flotantes (o números de punto flotante).

>>> 1 / 2

0.5

>>> 1 / 1

1.0

If you’d rather discard the fractional part and do integer division, you can use a double slash.

Si prefiere descartar la parte fraccionaria y hacer una división entera, puede usar una barra doble.

>>> 1 // 2

0

>>> 1 // 1

1

>>> 5.0 // 2.4

2.0

In older versions of Python, ordinary division on integers used to work like this double slash. If you’re using Python 2.x, you can get proper division by adding the following statement to the beginning of your program (writing full programs is described later) or simply executing it in the interactive interpreter:

En versiones anteriores de Python, la división ordinaria en números enteros solía funcionar como esta doble barra. Si está utilizando Python 2.x, puede obtener una división adecuada agregando la siguiente declaración al comienzo de su programa (la escritura de programas completos se describe más adelante) o simplemente ejecutándola en el intérprete interactivo:

>>> from \_\_future\_\_ import division

Note In case it’s not entirely clear, the future in the instruction is surrounded by two underscores on both sides: \_ \_future\_ \_.

Nota En caso de que no quede del todo claro, el futuro en la instrucción está rodeado por dos guiones bajos en ambos lados: \_ \_futuro\_ \_.

Another alternative, if you’re running an old Python from the command line, is to supply the command- line switch -Qnew. There is a more thorough explanation of the \_\_future\_\_ stuff in the section “Back to the \_\_future\_\_” later in this chapter.

Otra alternativa, si está ejecutando un Python antiguo desde la línea de comandos, es proporcionar el modificador de línea de comandos -Qnew. Hay una explicación más completa de las cosas del \_\_futuro\_\_ en la sección “Regreso al \_\_futuro\_\_” más adelante en este capítulo.

Now you’ve seen the basic arithmetic operators (addition, subtraction, multiplication, and division), but I’ve left out a close relative of integer division.

Ahora que has visto los operadores aritméticos básicos (suma, resta, multiplicación y división), pero he omitido un pariente cercano de la división de enteros.

>>> 1 % 2

1

This is the remainder (modulus) operator. x % y gives the remainder of x divided by y. In other words, it’s the part that’s left over when you use integer division. That is, x % y is the same as x - ((x // y) \* y).

Este es el operador de resto (módulo). x % y da el resto de x dividido por y. En otras palabras, es la parte que sobra cuando usas la división de enteros. Es decir, x % y es lo mismo que x - ((x // y) \* y).

>>> 10 // 3

3

>>> 10 % 3

1

>>> 9 // 3

3

>>> 9 % 3

0

>>> 2.75 % 0.5

0.25

Here 10 // 3 is 3 because the result is rounded down. But 3 × 3 is 9, so you get a remainder of 1. When you divide 9 by 3, the result is exactly 3, with no rounding. Therefore, the remainder is 0. This may be useful if you want to check something “every 10 minutes” as in the recipe earlier in the chapter. You can simply check whether minute % 10 is 0. (For a description on how to do this, see the sidebar “Sneak Peek: The if Statement” later in this chapter.) As you can see from the final example, the remainder operator works just fine with floats as well. It even works with negative numbers, and this can be a little confusing.

Aquí 10 // 3 es 3 porque el resultado se redondea hacia abajo. Pero 3 × 3 es 9, por lo que obtienes un resto de 1. Cuando divides 9 entre 3, el resultado es exactamente 3, sin redondeo. Por lo tanto, el resto es 0. Esto puede ser útil si desea verificar algo "cada 10 minutos" como en la receta anterior en el capítulo. Simplemente puede verificar si el minuto % 10 es 0. (Para obtener una descripción sobre cómo hacer esto, consulte la barra lateral "Adelanto: la declaración if" más adelante en este capítulo). Como puede ver en el ejemplo final, el operador de resto funciona bien con flotadores también. Incluso funciona con números negativos, y esto puede ser un poco confuso.

>>> 10 % 3

1

>>> 10 % -3

-2

>>> -10 % 3

2

>>> -10 % -3

-1

Looking at these examples, it might not be immediately obvious how it works. It’s probably easier to understand if you look at the companion operation of integer division.

Mirando estos ejemplos, puede que no sea inmediatamente obvio cómo funciona. Probablemente sea más fácil de entender si observa la operación complementaria de la división de enteros.

>>> 10 // 3

3

>>> 10 // -3

-4

>>> -10 // 3

-4

>>> -10 // -3

3

Given how the division works, it’s not that hard to understand what the remainder must be. The important thing to understand about integer division is that it is rounded down, which for negative numbers is away from zero. That means -10 // 3 is rounded down to -4, not up to -3. The last operator we’ll look at is the exponentiation (or power) operator.

Dado cómo funciona la división, no es tan difícil entender cuál debe ser el resto. Lo importante que debe entender acerca de la división de enteros es que se redondea hacia abajo, lo que para los números negativos está lejos de cero. Eso significa que -10 // 3 se redondea a -4, no a -3. El último operador que veremos es el operador de exponenciación (o potencia).

>>> 2 \*\* 3

8

>>> -3 \*\* 2

-9

>>> (-3) \*\* 2

9

Note that the exponentiation operator binds tighter than the negation (unary minus), so -3\*\*2 is in fact the same as -(3\*\*2). If you want to calculate (-3)\*\*2, you must say so explicitly.

Tenga en cuenta que el operador de exponenciación se vincula más estrechamente que la negación (menos unario), por lo que -3\*\*2 es de hecho lo mismo que -(3\*\*2). Si desea calcular (-3)\*\*2, debe decirlo explícitamente.

Hexadecimals Octals and Binary

To conclude this section, I should mention that hexadecimal, octal, and binary numbers are written like this:

Para concluir esta sección, debo mencionar que los números hexadecimales, octales y binarios se escriben así:

>>> 0xAF

175

>>> 010

8

>>> 0b1011010010

722

The first digit in both of these is zero. (If you don’t know what this is all about, you probably don’t need this quite yet. Just file it away for later use.)

El primer dígito en ambos es cero. (Si no sabe de qué se trata, probablemente no lo necesite todavía. Simplemente archívelo para usarlo más tarde).

Variables

Another concept that might be familiar to you is variables. If algebra is but a distant memory, don’t worry: variables in Python are easy to understand. A variable is a name that represents (or refers to) some value. For example, you might want the name x to represent 3. To make it so, simply execute the following:

Otro concepto que puede resultarle familiar es el de las variables. Si el álgebra es solo un recuerdo lejano, no se preocupe: las variables en Python son fáciles de entender. Una variable es un nombre que representa (o se refiere a) algún valor. Por ejemplo, es posible que desee que el nombre x represente 3. Para hacerlo, simplemente ejecute lo siguiente:

>>> x = 3

This is called an assignment. We assign the value 3 to the variable x. Another way of putting this is to say that we bind the variable x to the value (or object) 3. After you’ve assigned a value to a variable, you can use the variable in expressions.

Esto se llama una asignación. Asignamos el valor 3 a la variable x. Otra forma de expresar esto es decir que vinculamos la variable x al valor (u objeto) 3. Después de haber asignado un valor a una variable, puede usar la variable en expresiones.

>>> x \* 2

6

Unlike some other languages, you can’t use a variable before you bind it to something. There is no “default value.”

A diferencia de otros lenguajes, no puede usar una variable antes de vincularla a algo. No existe un valor predeterminado."

Note the simple story is that names, or identifiers, in python consist of letters, digits, and underscore characters (\_). they can’t begin with a digit, so Plan9 is a valid variable name, whereas 9Plan is not.3

Tenga en cuenta que la historia simple es que los nombres, o identificadores, en python consisten en letras, dígitos y caracteres de subrayado (\_). no pueden comenzar con un dígito, por lo que Plan9 es un nombre de variable válido, mientras que 9Plan no lo es.3

Statements

Until now we’ve been working (almost) exclusively with expressions, the ingredients of the recipe. But what about statements—the instructions? In fact, I’ve cheated. I’ve introduced two types of statements already: the print statement and assignments. What’s the difference between a statement and an expression? You could think of it like this: an expression is something, while a statement does something. For example, 2 \* 2 is 4, whereas print(2 \* 2) prints 4. The two behave quite similarly, so the difference between them might not be all that clear.

Hasta ahora hemos estado trabajando (casi) exclusivamente con expresiones, los ingredientes de la receta. Pero, ¿qué pasa con las declaraciones, las instrucciones? De hecho, he hecho trampa. Ya he introducido dos tipos de declaraciones: la declaración de impresión y las asignaciones. ¿Cuál es la diferencia entre un enunciado y una expresión? Podría pensarlo así: una expresión es algo, mientras que una declaración hace algo. Por ejemplo, 2 \* 2 es 4, mientras que print(2 \* 2) imprime 4. Los dos se comportan de manera bastante similar, por lo que la diferencia entre ellos podría no ser tan clara.

>>> 2 \* 2

4

>>> print(2 \* 2)

4

As long as you execute this in the interactive interpreter, there’s no difference, but that is only because the interpreter always prints out the values of all expressions (using the same representation as repr—see the section “String Representations, str and repr” later in this chapter). That is not true of Python in general. Later in this chapter, you’ll see how to make programs that run without this interactive prompt; simply putting an expression such as 2 \* 2 in your program won’t do anything interesting.4 Putting print(2 \* 2) in there, however, will still print out 4.

Mientras ejecute esto en el intérprete interactivo, no hay diferencia, pero eso es solo porque el intérprete siempre imprime los valores de todas las expresiones (usando la misma representación que repr; consulte la sección "Representaciones de cadenas, str y repr" más adelante en este capítulo). Eso no es cierto para Python en general. Más adelante en este capítulo, verá cómo crear programas que se ejecuten sin este aviso interactivo; simplemente poner una expresión como 2 \* 2 en su programa no hará nada interesante.4 Poner print(2 \* 2) allí, sin embargo, seguirá imprimiendo 4.

Note actually, print is a function (more on those later in the chapter), so what I’m referring to as a print statement is simply a function call. In python 2.x, print had a statement type of its own and didn’t use parentheses around its arguments.

Tenga en cuenta que, en realidad, imprimir es una función (más sobre esto más adelante en el capítulo), por lo que me estoy refiriendo a una declaración de impresión es simplemente una llamada de función. En python 2.x, print tenía un tipo de declaración propio y no usaba paréntesis alrededor de sus argumentos.

The difference between statements and expressions is more obvious when dealing with assignments. Because they are not expressions, they have no values that can be printed out by the interactive interpreter.

>>> x = 3

>>>

You simply get a new prompt immediately. Something has changed, however. We now have a new variable x, which is now bound to the value 3. To some extent, this is a defining quality of statements in general: they change things. For example, assignments change variables, and print statements change how your screen looks.

3 The slightly less simple story is that the rules for identifier names are in part based on the Unicode standard, as documented in the Python Language Reference at https://docs.python.org/3/reference/lexical\_analysis.html.

4 In case you’re wondering—yes, it does do something. It calculates the product of 2 and 2. However, the result isn’t kept anywhere or shown to the user; it has no side effects, beyond the calculation itself

Assignments are probably the most important type of statement in any programming language, although it may be difficult to grasp their importance right now. Variables may just seem like temporary “storage” (like the pots and pans of a cooking recipe), but the real power of variables is that you don’t need to know what values they hold in order to manipulate them. For example, you know that x \* y evaluates to the product of x and y, even though you may have no knowledge of what x and y are. So, you may write programs that use variables in various ways without knowing the values they will eventually hold (or refer to) when the program is run.

Getting Input from the User

You’ve seen that you can write programs with variables without knowing their values. Of course, the interpreter must know the values eventually. So how can it be that we don’t? The interpreter knows only what we tell it, right? Not necessarily. You may have written a program, and someone else may use it. You cannot predict what values users will supply to the program. Let’s take a look at the useful function input. (I’ll have more to say about functions in a minute.)

>>> input("The meaning of life: ")

The meaning of life: 42

'42'

What happens here is that the first line (input(...)) is executed in the interactive interpreter. It prints out the string "The meaning of life: " as a new prompt. I type 42 and press Enter. The resulting value of input is that very number (as a piece of text, or string), which is automatically printed out in the last line. Converting the strings to integers using int, we can construct a slightly more interesting example:

>>> x = input("x: ")

x: 34

>>> y = input("y: ")

y: 42

>>> print(int(x) \* int(y))

1428

Here, the statements at the Python prompts (>>>) be part of a finished program, and the values entered (34 and 42) would be supplied by some user. Your program would then print out the value 1428, which is the product of the two. And you didn’t have to know these values when you wrote the program, right?

Note getting input like this is much more useful when you save your programs in a separate file so other users can execute them. You learn how to do that later in this chapter, in the section “saving and executing Your programs.”

SNEAK PEEK: THE IF STATEMENT

to spice things up a bit, I’ll give you a sneak peek of something you aren’t really supposed to learn about until Chapter 5: the if statement. the if statement lets you perform an action (another statement) if a given condition is true. One type of condition is an equality test, using the equality operator, ==. Yes, it’s a double equality sign. (the single one is used for assignments, remember?)

You put this condition after the word if and then separate it from the following statement with a colon.

>>> if 1 == 2: print('One equals two')

...

>>> if 1 == 1: print('One equals one')

...

One equals one

>>>

nothing happens when the condition is false. When it is true, however, the statement following the colon (in this case, a print statement) is executed. note also that when using if statements in the interactive interpreter, you need to press enter twice before it is executed. (the reason for this will become clear in Chapter 5.) so, if the variable time is bound to the current time in minutes, you could check whether you’re “on the hour” with the following statement:

if time % 60 == 0: print('On the hour!')

Functions

In the “Numbers and Expressions” section, I used the exponentiation operator (\*\*) to calculate powers. The fact is that you can use a function instead, called pow.

>>> 2 \*\* 3

8

>>> pow(2, 3)

8

A function is like a little program that you can use to perform a specific action. Python has a lot of functions that can do many wonderful things. In fact, you can make your own functions, too (more about that later); therefore, we often refer to standard functions such as pow as built-in functions. Using a function as I did in the preceding example is called calling the function. You supply it with arguments (in this case, 2 and 3), and it returns a value to you. Because it returns a value, a function call is simply another type of expression, like the arithmetic expressions discussed earlier in this chapter.6 In fact, you can combine function calls and operators to create more complicated expressions (like I did with int, earlier).

>>> 10 + pow(2, 3 \* 5) / 3.0

10932.666666666666

Several built-in functions can be used in numeric expressions like this. For example, abs gives the absolute value of a number, and round rounds floating-point numbers to the nearest integer.

>>> abs(-10)

10

>>> 2 // 3

0

>>> round(2 / 3)

1.0

Notice the difference between the two last expressions. Integer division always rounds down, whereas round rounds to the nearest integer, with ties rounded toward the even number. But what if you want to round a given number down? For example, you might know that a person is 32.9 years old, but you would like to round that down to 32 because she isn’t really 33 yet. Python has a function for this (called floor)—it just isn’t available directly. As is the case with many useful functions, it is found in a module.

Modules

You may think of modules as extensions that can be imported into Python to expand its capabilities. You import modules with a special command called (naturally enough) import. The function mentioned in the previous section, floor, is in a module called math.

>>> import math

>>> math.floor(32.9)

32

Notice how this works: we import a module with import and then use the functions from that module by writing module.function. For this operation in particular, you could actually just convert the number into an integer, like I did earlier, with the results from input.

>>> int(32.9)

32

Note similar functions exist to convert to other types (for example, str and float). In fact, these aren’t really functions—they’re classes. I’ll have more to say about classes later.

The math module has several other useful functions, though. For example, the opposite of floor is ceil (short for “ceiling”), which finds the smallest integral value larger than or equal to the given number.

>>> math.ceil(32.3)

33

>>> math.ceil(32)

32

If you are sure that you won’t import more than one function with a given name (from different modules), you might not want to write the module name each time you call the function. Then you can use a variant of the import command.

>>> from math import sqrt

>>> sqrt(9)

3.0

After using the from module import function, you can use the function without its module prefix.

Tip You may, in fact, use variables to refer to functions (and most other things in python). By performing the assignment foo = math.sqrt, you can start using foo to calculate square roots; for example, foo(4) yields 2.0.

cmath and Complex Numbers

The sqrt function is used to calculate the square root of a number. Let’s see what happens if we supply it with a negative number:

>>> from math import sqrt

>>> sqrt(-1)

Traceback (most recent call last):

...

ValueError: math domain error

or, on some platforms:

>>> sqrt(-1)

nan

Note nan is simply a special value meaning “not a number.”

If we restrict ourselves to real numbers and their approximate implementation in the form of floats, we can’t take the square root of a negative number. The square root of a negative number is a so-called imaginary number, and numbers that are the sum of a real and an imaginary part are called complex. The Python standard library has a separate module for dealing with complex numbers.

>>> import cmath

>>> cmath.sqrt(-1)

1j

Notice that I didn’t use from ... import ... here. If I had, I would have lost my ordinary sqrt. Name clashes like these can be sneaky, so unless you really want to use the from version, you should probably stick with a plain import.

The value 1j is an example of an imaginary number. These numbers are written with a trailing j (or J). Complex arithmetic essentially follows from defining 1j as the square root of -1. Without delving too deeply into the topic, let me just show a final example:

>>> (1 + 3j) \* (9 + 4j)

(-3 + 31j)

As you can see, the support for complex numbers is built into the language.

Note there is no separate type for imaginary numbers in python. they are treated as complex numbers whose real component is zero.

Back to the \_\_future\_\_

It has been rumored that Guido van Rossum (Python’s creator) has a time machine—on more than one occasion when people have requested features in the language, they have found that the features were already implemented. Of course, we aren’t all allowed into this time machine, but Guido has been kind enough to build a part of it into Python, in the form of the magic module \_\_future\_\_. From it, we can import features that will be standard in Python in the future but that aren’t part of the language yet. You saw this in the “Numbers and Expressions” section, and you’ll be bumping into it from time to time throughout this book.

Saving and Executing Your Programs

The interactive interpreter is one of Python’s great strengths. It makes it possible to test solutions and to experiment with the language in real time. If you want to know how something works, just try it! However, everything you write in the interactive interpreter is lost when you quit. What you really want to do is write programs that both you and other people can run. In this section, you learn how to do just that. First of all, you need a text editor, preferably one intended for programming. (If you use something like Microsoft Word, which I really don’t really recommend, be sure to save your code as plain text.) If you are already using IDLE, you’re in luck. With IDLE, you can simply create a new editor window with File › New File. Another window appears, without an interactive prompt. Whew! Start by entering the following:

print("Hello, world!")

Now select File › Save to save your program (which is, in fact, a plain text file). Be sure to put it somewhere where you can find it later, and give your file any reasonable name, such as hello.py. (The .py ending is significant.)

Got that? Don’t close the window with your program in it. If you did, just open it again (File › Open). Now you can run it with Run › Run Module. (If you aren’t using IDLE, see the next section about running your programs from the command prompt.) What happens? Hello, world! is printed in the interpreter window, which is exactly what we wanted. The interpreter prompt may be gone (depending on the version you’re using), but you can get it back by pressing Enter (in the interpreter window). Let’s extend our script to the following:

name = input("What is your name? ")

print("Hello, " + name + "!")

p11

If you run this (remember to save it first), you should see the following prompt in the interpreter window:

What is your name?

Enter your name (for example, Gumby) and press Enter. You should get something like this:

Hello, Gumby!

TURTLE POWER!

the print statement is useful for basic examples because it works virtually everywhere. If you’d like to experiment with more visually interesting output, you should take a look at the turtle module, which implements so-called turtle graphics. If you have IDLe up and running, the turtle module should work just fine, and it lets you draw figures rather than print text. though it is a practice you should be wary of in general, while playing around with turtle graphics, it can be convenient to simply import all names from the module.

from turtle import \*

Once you’ve figured out which functions you need, you can go back to only importing those. the idea of turtle graphics stems from actual turtle-like robots that could move forward and backward and turn a given number of degrees left or right. In addition, they carried a pen, which they could move up or down to determine whether it touched the piece of paper they were moving on. the turtle module gives you a simulation of such a robot. For example, here’s how you’d draw a triangle:

forward(100)

left(120)

forward(100)

left(120)

forward(100)

If you run this, a new window should appear, with a little arrow-shaped “turtle” moving around, with a line trailing behind it. to ask it to lift the pen, you use penup(), and to put it down again, pendown(). For more commands, consult the relevant section of the python Library reference (https://docs.python.org/3/library/turtle.html), and for drawing ideas, try a web search for turtle graphics. as you learn additional concepts, you might want to experiment with turtle alternatives to the more mundane print examples. and playing around with turtle graphics quickly demonstrates the need for some of the basic programming constructs I’ll be showing you. (For example, how would you avoid repeating the forward and left commands in the previous example? how would you draw, say, an octagon instead of a triangle? Or several regular polygons with different number of sides, with as few lines of code as possible?)

Running Your Python Scripts from a Command Prompt

Actually, there are several ways to run your programs. First, let’s assume you have a DOS window or a UNIX shell prompt before you and that the directory containing the Python executable (called python.exe in Windows, and python in UNIX) or the directory containing the executable (in Windows) has been put in your PATH environment variable. 7 Also, let’s assume that your script from the previous section (hello.py) is in the current directory. Then you can execute your script with the following command in Windows:

C:\>python hello.py

or UNIX:

$ python hello.py

As you can see, the command is the same. Only the system prompt changes.

Making Your Scripts Behave Like Normal Programs

Sometimes you want to execute a Python program (also called a script) the same way you execute other programs (such as your web browser or text editor), rather than explicitly using the Python interpreter. In UNIX, there is a standard way of doing this: have the first line of your script begin with the character sequence #! (called pound bang or shebang) followed by the absolute path to the program that interprets the script (in our case Python). Even if you didn’t quite understand that, just put the following in the first line of your script if you want it to run easily on UNIX:

#!/usr/bin/env python

This should run the script, regardless of where the Python binary is located. If you have more than one version of Python installed, you could use a more specific executable name, such as python3, rather than simply python.

Before you can actually run your script, you must make it executable.

$ chmod a+x hello.py

Now it can be run like this (assuming that you have the current directory in your path):

$ hello.py

If this doesn’t work, try using ./hello.py instead, which will work even if the current directory (.) is not part of your execution path (which a responsible sysadmin would probably tell you it shouldn’t be). If you like, you can rename your file and remove the py suffix to make it look more like a normal

program.

What About Double-Clicking?

In Windows, the suffix (.py) is the key to making your script behave like a program. Try double-clicking the file hello.py you saved in the previous section. If Python was installed correctly, a DOS window appears with the prompt “What is your name?”8 There is one problem with running your program like this, however. Once you’ve entered your name, the program window closes before you can read the result. The window closes when the program is finished. Try changing the script by adding the following line at the end:

input("Press <enter>")

Now, after running the program and entering your name, you should have a DOS window with the following contents:

What is your name? Gumby

Hello, Gumby!

Press <enter>

Once you press the Enter key, the window closes (because the program is finished).

Comments

The hash sign (#) is a bit special in Python. When you put it in your code, everything to the right of it is ignored (which is why the Python interpreter didn’t choke on the /usr/bin/env stuff used earlier). Here is an example:

# Print the circumference of the circle:

print(2 \* pi \* radius)

The first line here is called a comment, which can be useful in making programs easier to understand—

both for other people and for yourself when you come back to old code. It has been said that the first

commandment of programmers is “Thou Shalt Comment” (although some less charitable programmers

swear by the motto “If it was hard to write, it should be hard to read”). Make sure your comments say

significant things and don’t simply restate what is already obvious from the code. Useless, redundant

comments may be worse than none. For example, in the following, a comment isn’t really called for:

# Get the user's name:

user\_name = input("What is your name?")

It’s always a good idea to make your code readable on its own as well, even without the comments. Luckily, Python is an excellent language for writing readable programs

Strings

Now what was all that "Hello, " + name + "!" stuff about? The first program in this chapter was simply

print("Hello, world!")

It is customary to begin with a program like this in programming tutorials. The problem is that I haven’t really explained how it works yet. You know the basics of the print statement (I’ll have more to say about that later), but what is "Hello, world!"? It’s called a string (as in “a string of characters”). Strings are found in almost every useful, real-world Python program and have many uses. Their main use is to represent bits of text, such as the exclamation “Hello, world!”

Single-Quoted Strings and Escaping Quotes

Strings are values, just as numbers are:

>>> "Hello, world!"

'Hello, world!'

There is one thing that may be a bit surprising about this example, though: when Python printed out our string, it used single quotes, whereas we used double quotes. What’s the difference? Actually, there is no difference.

>>> 'Hello, world!'

'Hello, world!'

Here, we use single quotes, and the result is the same. So why allow both? Because in some cases it may be useful.

>>> "Let's go!"

"Let's go!"

>>> '"Hello, world!" she said'

'"Hello, world!" she said'

In the preceding code, the first string contains a single quote (or an apostrophe, as we should perhaps call it in this context), and therefore we can’t use single quotes to enclose the string. If we did, the interpreter would complain (and rightly so).

>>> 'Let's go!'

SyntaxError: invalid syntax

Here, the string is 'Let', and Python doesn’t quite know what to do with the following s (or the rest of the line, for that matter). In the second string, we use double quotes as part of our sentence. Therefore, we have to use single quotes to enclose our string, for the same reasons as stated previously. Or, actually we don’t have to. It’s just convenient. An alternative is to use the backslash character (\) to escape the quotes in the string, like this:

>>> 'Let\'s go!'

"Let's go!"

Python understands that the middle single quote is a character in the string and not the end of the string. (Even so, Python chooses to use double quotes when printing out the string.) The same works with double quotes, as you might expect.

>>> "\"Hello, world!\" she said"

'"Hello, world!" she said'

Escaping quotes like this can be useful, and sometimes necessary. For example, what would you do without the backslash if your string contained both single and double quotes, as in the string 'Let\'s say "Hello, world!"'?

Note tired of backslashes? as you will see later in this chapter, you can avoid most of them by using long strings and raw strings (which can be combined).

Concatenating Strings

Just to keep whipping this slightly tortured example, let me show you another way of writing the same string:

>>> "Let's say " '"Hello, world!"'

'Let\'s say "Hello, world!"'

I’ve simply written two strings, one after the other, and Python automatically concatenates them (makes them into one string). This mechanism isn’t used very often, but it can be useful at times. However, it works only when you actually write both strings at the same time, directly following one another.

>>> x = "Hello, "

>>> y = "world!"

>>> x y

SyntaxError: invalid syntax

In other words, this is just a special way of writing strings, not a general method of concatenating them. How, then, do you concatenate strings? Just like you add numbers:

>>> "Hello, " + "world!"

'Hello, world!'

>>> x = "Hello, "

>>> y = "world!"

>>> x + y

'Hello, world!'

String Representations, str and repr

Throughout these examples, you have probably noticed that all the strings printed out by Python are still quoted. That’s because it prints out the value as it might be written in Python code, not how you would like it to look for the user. If you use print, however, the result is different.

>>> "Hello, world!"

'Hello, world!'

>>> print("Hello, world!")

Hello, world!

The difference is even more obvious if we sneak in the special linefeed character code \n.

>>> "Hello,\norld!"

'Hello,\nworld!'

>>> print("Hello,\nworld!")

Hello,

world!

Values are converted to strings through two different mechanisms. You can access both mechanisms yourself, by using the functions str and repr.9 With str, you convert a value into a string in some reasonable fashion that will probably be understood by a user, for example, converting any special character codes to the corresponding characters, where possible. If you use repr, however, you will generally get a representation of the value as a legal Python expression.

>>> print(repr("Hello,\nworld!"))

'Hello,\nworld!'

>>> print(str("Hello,\nworld!"))

Hello,

world!

Long Strings, Raw Strings, and bytes

There are some useful, slightly specialized ways of writing strings. For example, there’s a custom syntax for writing strings that include newlines (long strings) or backslashes (raw strings). In Python 2, there was also a separate syntax for writing strings with special symbols of different kinds, producing objects of the unicode type. The syntax still works but is now redundant, because all strings in Python 3 are Unicode strings. Instead, a new syntax has been introduced to specify a bytes object, roughly corresponding to the old- school strings. As we shall see, these still play an important part in the handling of Unicode encodings.

Long Strings

If you want to write a really long string, one that spans several lines, you can use triple quotes instead of ordinary quotes.

print('''This is a very long string. It continues here.

And it's not over yet. "Hello, world!"

Still here.''')

You can also use triple double quotes, """like this""". Note that because of the distinctive enclosing quotes, both single and double quotes are allowed inside, without being backslash-escaped.

Tip Ordinary strings can also span several lines. If the last character on a line is a backslash, the line break itself is “escaped” and ignored. For example:

print("Hello, \ world!")

would print out Hello, world!. the same goes for expressions and statements in general.

>>> 1 + 2 + \

4 + 5

12

>>> print \

('Hello, world')

Hello, world

Raw Strings

Raw strings aren’t too picky about backslashes, which can be very useful sometimes.10 In ordinary strings, the backslash has a special role: it escapes things, letting you put things into your string that you couldn’t normally write directly. For example, as we’ve seen, a newline is written \n and can be put into a string like this:

>>> print('Hello,\nworld!')

Hello,

world!

This is normally just dandy, but in some cases, it’s not what you want. What if you wanted the string to include a backslash followed by an n? You might want to put the DOS pathname C:\nowhere into a string.

>>> path = 'C:\nowhere'

>>> path

'C:\nowhere'

This looks correct, until you print it and discover the flaw.

>>> print(path)

C:

owhere

It’s not exactly what we were after, is it? So what do we do? We can escape the backslash itself.

>>> print('C:\\nowhere')

C:\nowhere

This is just fine. But for long paths, you wind up with a lot of backslashes.

path = 'C:\\Program Files\\fnord\\foo\\bar\\baz\\frozz\\bozz'

Raw strings are useful in such cases. They don’t treat the backslash as a special character at all. Every

character you put into a raw string stays the way you wrote it.

>>> print(r'C:\nowhere')

C:\nowhere

>>> print(r'C:\Program Files\fnord\foo\bar\baz\frozz\bozz')

C:\Program Files\fnord\foo\bar\baz\frozz\bozz

As you can see, raw strings are prefixed with an r. It would seem that you can put anything inside a raw string, and that is almost true. Quotes must be escaped as usual, although that means you get a backslash in your final string, too.

>>> print(r'Let\'s go!')

Let\'s go!

The one thing you can’t have in a raw string is a lone, final backslash. In other words, the last character in a raw string cannot be a backslash unless you escape it (and then the backslash you use to escape it will be part of the string, too). Given the previous example, that ought to be obvious. If the last character (before the final quote) is an unescaped backslash, Python won’t know whether or not to end the string.

>>> print(r"This is illegal\")

SyntaxError: EOL while scanning string literal

Okay, so it’s reasonable, but what if you want the last character in your raw string to be a backslash? (Perhaps it’s the end of a DOS path, for example.) Well, I’ve given you a whole bag of tricks in this section that should help you solve that problem, but basically you need to put the backslash in a separate string. A simple way of doing that is the following:

>>> print(r'C:\Program Files\foo\bar' '\\')

C:\Program Files\foo\bar\

Note that you can use both single and double quotes with raw strings. Even triple-quoted strings can be raw.

Unicode, bytes, and bytearray

Python strings represent text using a scheme known as Unicode. The way this works for most basic programs is pretty transparent, so if you’d like, you could skip this section for now and read up on the topic as needed. However, as string and text file handling is one of the main uses of Python code, it probably wouldn’t hurt to at least skim this section. Abstractly, each Unicode character is represented by a so-called code point, which is simply its number in the Unicode standard. This allows you to refer to more than 120,000 characters in 129 writing systems in a way that should be recognizable by any modern software. Of course, your keyboard won’t have hundreds of thousands of keys, so there are general mechanisms for specifying Unicode characters, either by 16- or 32-bit hexadecimal literals (prefixing them with \u or \U, respectively) or by their Unicode name (using \N{name}).

>>> "\u00C6"

'Æ'

>>> "\U0001F60A"

''

>>> "This is a cat: \N{Cat}"

'This is a cat: '

You can find the various code points and names by searching the Web, using a description of the character you need, or you can use a specific site such as <http://unicode-table.com>

The idea of Unicode is quite simple, but it comes with some challenges, one of which is the issue

of encoding. All objects are represented in memory or on disk as a series of binary digits—zeroes and

ones—grouped in chunks of eight, or bytes, and strings are no exception. In programming languages such as C, these bytes are completely out in the open. Strings are simply sequences of bytes. To interoperate with C, for example, and to write text to files or send it through network sockets, Python has two similar types, the immutable bytes and the mutable bytearray. If you wanted, you could produce a bytes object directly, instead of a string, by using the prefix b:

>>> b'Hello, world!'

b'Hello, world!'

However, a byte can hold only 256 values, quite a bit less than what the Unicode standard requires. Python bytes literals permit only the 128 characters of the ASCII standard, with the remaining 128 byte values requiring escape sequences like \xf0 for the hexadecimal value 0xf0 (that is, 240).

It might seem the only difference here is the size of the alphabet available to us. That’s not really

accurate, however. At a glance, it might seem like both ASCII and Unicode refer to a mapping between non-negative integers and characters, but there is a subtle difference: where Unicode code points are defined as integers, ASCII characters are defined both by their number and by their binary encoding. One reason this seems completely unremarkable is that the mapping between the integers 0–255 and an eight-digit binary numeral is completely standard, and there is little room to maneuver. The thing is, once we go beyond the single byte, things aren’t that simple. The direct generalization of simply representing each code point as the corresponding binary numeral may not be the way to go. Not only is there the issue of byte order, which one bumps up against even when encoding integer values, there is also the issue of wasted space: if we use the same number of bytes for encoding each code point, all text will have to accommodate the fact that you might want to include a few Anatolian hieroglyphs or a smattering of Imperial Aramaic. There is a standard for such an encoding of Unicode, which is called UTF-32 (for Unicode Transformation Format 32 bits), but if you’re mainly handling text in one of the more common languages of the Internet, for example, this is quite wasteful.

There is an absolutely brilliant alternative, however, devised in large part by computing pioneer Kenneth Thompson. Instead of using the full 32 bits, it uses a variable encoding, with fewer bytes for some scripts than others. Assuming that you’ll use these scripts more often, this will save you space overall, similar to how Morse code saves you effort by using fewer dots and dashes for the more common letters. In particular, the ASCII encoding is still used for single-byte encoding, retaining compatibility with older systems. However, characters outside this range use multiple bytes (up to six). Let’s try to encode a string into bytes, using the ASCII, UTF-8, and UTF-32 encodings.

>>> "Hello, world!".encode("ASCII")

b'Hello, world!'

>>> "Hello, world!".encode("UTF-8")

b'Hello, world!'

>>> "Hello, world!".encode("UTF-32")

b'\xff\xfe\x00\x00H\x00\x00\x00e\x00\x00\x00l\x00\x00\x00l\x00\x00\x00o\x00\x00\x00,\x00\

x00\x00 \x00\x00\x00w\x00\x00\x00o\x00\x00\x00r\x00\x00\x00l\x00\x00\x00d\x00\x00\x00!\x00\

x00\x00'

As you can see, the first two are equivalent, while the last one is quite a bit longer. Here’s another example:

>>> len("How long is this?".encode("UTF-8"))

17

>>> len("How long is this?".encode("UTF-32"))

72

The difference between ASCII and UTF-8 appears once we use some slightly more exotic characters:

>>> "Hællå, wørld!".encode("ASCII")

Traceback (most recent call last):

...

UnicodeEncodeError: 'ascii' codec can't encode character '\xe6' in position 1: ordinal not

in range(128)

The Scandinavian letters here have no encoding in ASCII. If we really need ASCII encoding (which can certainly happen), we can supply another argument to encode, telling it what to do with errors. The normal mode here is 'strict', but there are others you can use to ignore or replace the offending characters.

>>> "Hællå, wørld!".encode("ASCII", "ignore")

b'Hll, wrld!'

>>> "Hællå, wørld!".encode("ASCII", "replace")

b'H?ll?, w?rld!'

>>> "Hællå, wørld!".encode("ASCII", "backslashreplace")

b'H\\xe6ll\\xe5, w\\xf8rld!'

>>> "Hællå, wørld!".encode("ASCII", "xmlcharrefreplace")

b'H&#230;ll&#229;, w&#248;rld!'

In almost all cases, though, you’ll be better off using UTF-8, which is in fact even the default encoding.

>>> "Hællå, wørld!".encode()

b'H\xc3\xa6ll\xc3\xa5, w\xc3\xb8rld!'

This is slightly longer than for the "Hello, world!" example, whereas the UTF-32 encoding would be of exactly the same length in both cases. Just like strings can be encoded into bytes, bytes can be decoded into strings.

>>> b'H\xc3\xa6ll\xc3\xa5, w\xc3\xb8rld!'.decode()

'Hællå, wørld!'

As before, the default encoding is UTF-8. We can specify a different encoding, but if we use the wrong one, we’ll either get an error message or end up with a garbled string. The bytes object itself doesn’t know about encoding, so it’s your responsibility to keep track of which one you’ve used. Rather than using the encode and decode methods, you might want to simply construct the bytes and str (i.e., string) objects, as follows:

>>> bytes("Hællå, wørld!", encoding="utf-8")

b'H\xc3\xa6ll\xc3\xa5, w\xc3\xb8rld!'

>>> str(b'H\xc3\xa6ll\xc3\xa5, w\xc3\xb8rld!', encoding="utf-8")

'Hællå, wørld!'

Using this approach is a bit more general and works better if you don’t know exactly the class of the string- like or bytes-like objects you’re working with—and as a general rule, you shouldn’t be too strict about that.

One of the most important uses for encoding and decoding is when storing text in files on disk. However, Python’s mechanisms for reading and writing files normally do the work for you! As long as you’re okay with having your files in UTF-8 encoding, you don’t really need to worry about it. But if you end up seeing gibberish where you expected text, perhaps the file was actually in some other encoding, and then it can be useful to know a bit about what’s going on. If you’d like to know more about Unicode in Python, check out the HOWTO on the subject.12

Note Your source code is also encoded, and the default there is UtF-8 as well. If you want to use some other encoding (for example, if your text editor insists on saving as something other than UtF-8), you can specify the encoding with a special comment.

# -\*- coding: encoding name -\*-

replace encoding name with whatever encoding you’re using (uppercase or lowercase), such as utf-8 or, perhaps more likely, latin-1, for example.

Finally, we have bytearray, a mutable version of bytes. In a sense, it’s like a string where you can modify the characters—which you can’t do with a normal string. However, it’s really designed more to be used behind the scenes and isn’t exactly user-friendly if used as a string-alike. For example, to replace a character, you have to assign an int in the range 0...255 to it. So if you want to actually insert a character, you have to get its ordinal value, using ord.

>>> x = bytearray(b"Hello!")

>>> x[1] = ord(b"u")

>>> x

bytearray(b'Hullo!')

A Quick Summary

This chapter covered quite a bit of material. Let’s take a look at what you’ve learned before moving on.

**Algorithms**: An algorithm is a recipe telling you exactly how to perform a task. When you program a computer, you are essentially describing an algorithm in a language the computer can understand, such as Python. Such a machine-friendly description is called a program, and it mainly consists of expressions and statements.

**Expressions**: An expression is a part of a computer program that represents a value. For example, 2 + 2 is an expression, representing the value 4. Simple expressions are built from literal values (such as 2 or "Hello") by using operators (such as + or %) and functions (such as pow). More complicated expressions can be created by combining simpler expressions (e.g., (2 + 2) \* (3 - 1)). Expressions may also contain variables.

**Variables**: A variable is a name that represents a value. New values may be assigned to variables through assignments such as x = 2. An assignment is a kind of statement.

**Statements:** A statement is an instruction that tells the computer to do something. That may involve changing variables (through assignments), printing things to the screen (such as print("Hello, world!")), importing modules, or doing a host of other stuff.

**Functions**: Functions in Python work just like functions in mathematics: they may take some arguments, and they return a result. (They may actually do lots of interesting stuff before returning, as you will find out when you learn to write your own functions in Chapter 6.)

**Modules**: Modules are extensions that can be imported into Python to extend its capabilities. For example, several useful mathematical functions are available in the math module.

**Programs**: You have looked at the practicalities of writing, saving, and running

Python programs. Strings: Strings are really simple—they are just pieces of text, with characters

represented as Unicode code points. And yet there is a lot to know about them. In this chapter, you’ve seen many ways to write them, and in Chapter 3 you learn many ways of using them.

New Functions in This Chapter

Functions Description

abs(number) Returns the absolute value of a number.

bytes(string, encoding[, errors]) Encodes a given string, with the specified behavior for errors.

cmath.sqrt(number) Returns the square root; works with negative numbers.

float(object) Converts a string or number to a floating-point number.

help([object]) Offers interactive help.

input(prompt) Gets input from the user as a string.

int(object) Converts a string or number to an integer.

math.ceil(number) Returns the ceiling of a number as a float.

math.floor(number) Returns the floor of a number as a float.

math.sqrt(number) Returns the square root; doesn’t work with negative

numbers.

pow(x, y[, z]) Returns x to the power of y (modulo z).

print(object, ...) Prints out the arguments, separated by spaces.

repr(object) Returns a string representation of a value.

round(number[, ndigits]) Rounds a number to a given precision, with ties rounded to

the even number.

str(object) Converts a value to a string. If converting from bytes, you

may specify encoding and error behavior.

What Now?

Now that you know the basics of expressions, let’s move on to something a bit more advanced: data structures. Instead of dealing with simple values (such as numbers), you’ll see how to bunch them together in more complex structures, such as lists and dictionaries. In addition, you’ll take another close look at strings. In Chapter 5, you learn more about statements, and after that you’ll be ready to write some really nifty programs.