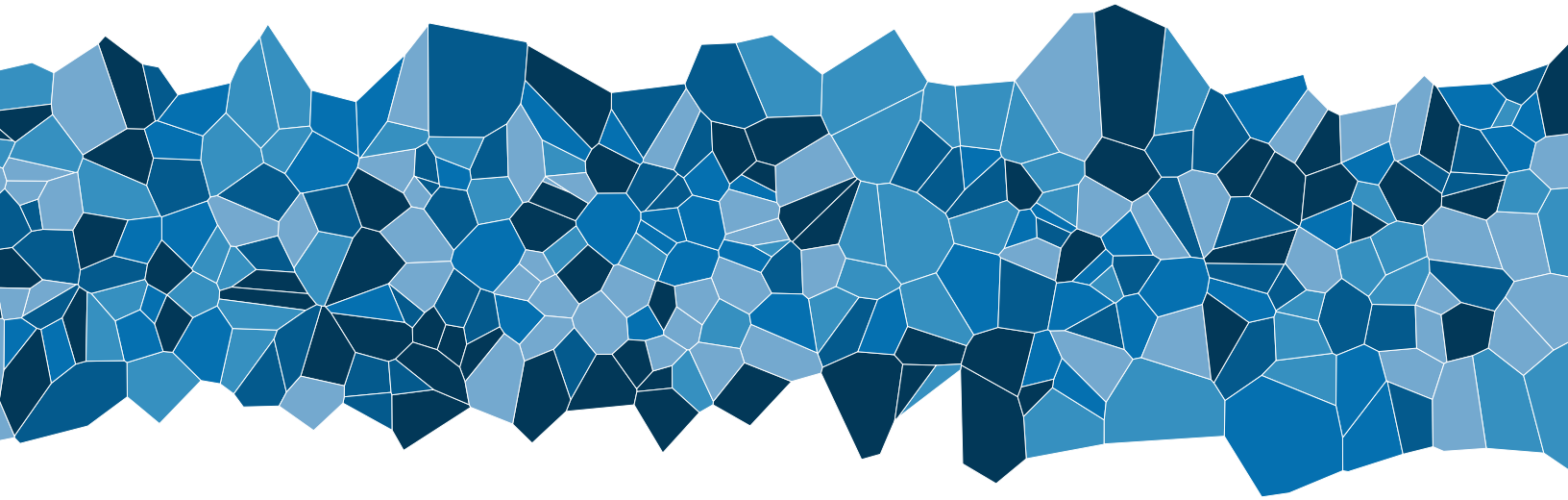
Ciencia de Datos para el Análisis Delictivo con Python

Andrew P. Wheeler



CRIME De-Coder

Ciencia de Datos para el Análisis Delictivo con Python

Andrew P. Wheeler

2025-10-25

Ciencia de Datos para el Análisis Delictivo con Python

ISBN 979-8-9903770-2-8 (Spanish ebook), ISBN 979-8-9903770-3-5 (Spanish paperback)

Echa un vistazo a mis otros trabajos en https://crimede-coder.com/

Índice

Preámbulo

Python es un lenguaje de programación de código abierto y gratuito, que permite escribir programas sencillos (o complejos) para que tu computadora realice tareas. Como breve ejemplo, aquí tienes un fragmento de código de Python que te indica la diferencia en el número de días entre dos fechas. Las líneas que empiezan con `#` son comentarios en Python; las otras líneas realizan distintas operaciones en código de Python. El texto en la parte gris de abajo es el código de Python, y el texto de la sección azul es la salida del programa.

```
# importing library to calculate times
from datetime import datetime

# creating two datetime objects
begin = datetime(2022,1,16)
end = datetime(2023,1,16)

# calculating the difference
dif = end - begin

# printing the result
print(dif.days)
```

```
365
```

Así que este es un programa trivial: podrías calcular el número de días entre las dos fechas usando varias herramientas. El poder de saber programar en Python es que puedes escribir un código para realizar (casi) cualquier cálculo que quieras. Un ejemplo común para un analista criminal puede ser consultar una base de datos y crear una tabla de conteos de delitos acumulados en lo que va del año para este año frente al anterior. Luego, puedes ejecutar el código cuando quieras, y este actualizará las estadísticas acumuladas en lo que va del año con la frecuencia que desees. En la práctica, un informe así no es más que encadenar pequeños ejemplos de código como los de arriba en series de operaciones más complejas.

¿Para quienes está dirigido este libro?

Este libro está dirigido a personas sin experiencia (o con experiencia inicial) en programación, pero que están interesadas en usar código para realizar análisis cuantitativos y automatizar tareas. El público principal al que va dirigido el libro está compuesto por analistas delictivos, pero cualquier persona que

busque iniciarse en la programación y el análisis de datos debería encontrar útil el contenido del libro. Además de los analistas delictivos, quienes deseen avanzar en su carrera en un rol de ciencia de datos o realizar investigación a nivel de posgrado (y que tengan formación en justicia penal) encontrarán útil el libro y sus ejemplos.

Hay muchos recursos actuales sobre el uso de python en internet – se puede usar un motor de búsqueda para encontrar diversos recursos completamente gratuitos en línea. Habitualmente escribo sobre computación técnica en andrewpwheeler.com, que es gratuito para que lo lea cualquiera. Sin embargo, estos recursos gratuitos a menudo son desordenados y resultan muy difíciles para que los principiantes los entiendan y se pongan en marcha. Cosas como "¿Cómo ejecuto un script sencillo de python?" o "¿Cómo instalo una biblioteca de python?" no son temas típicos que se cubran ni siquiera en materiales introductorios de python en línea. Este libro pretende constituir un recurso único para que las personas dedicadas al análisis delictivo puedan empezar.

Mi intención con este libro no es solo presentar ejemplos de código en python, sino también describir otros pasos necesarios para principiantes, como configurar entornos de python y automatizar tareas usando scripts de shell. No te preocupes si no entiendes qué son por el momento – ¡se explicarán!. Incluso dedico tiempo a describir una estructura típica de proyecto que es bastante estándar en el desarrollo de software más profesional. Estas son cosas que no están relacionadas directamente con la programación, pero son necesarias para poder empezar a usar python y utilizarlo de forma eficaz.

Así, este libro llena un nicho – una introducción a la realización de tareas en python relevantes para los analistas de delitos. El libro contiene lo siguiente:

- instalar y crear entornos de Python
- una introducción a la programación en Python
- trabajar con datos tabulares utilizando bibliotecas científicas
- usar SQL para consultar bases de datos
- automatizar la creación de informes y crear tablas y gráficos de alta calidad

Estos son los ingredientes necesarios, tanto en términos de programación como de proyectos más realistas, que permiten a una persona ser más productiva en sus tareas habituales con Python.

Lo que este libro no es

Al abordar el aprendizaje de la programación y el análisis de datos, muchos libros incluyen *ambos* al mismo tiempo. Esto suele ser un error, ya que puede aumentar considerablemente la carga para quienes desean aprender el material.

Este libro *no* está pensado como una introducción al análisis del crimen como tema en general. Para quienes deseen aprender estadística básica y los análisis que realizan los analistas del crimen, sugeriría consultar los materiales del curso en mi sitio web personal, así como los materiales de la Asociación Internacional de Analistas del Crimen (IACA). Si la demanda es suficiente, podría crear libros futuros para cubrir con mayor profundidad la estadística introductoria para analistas del crimen, ¡así que háganmelo saber si es algo que les interesa!

Este libro tiene como objetivo ayudarte a empezar a escribir códigos y aplicarlos a tareas reales que los analistas delictivos necesitan realizar. Utilizo ejemplos realistas que podrían interesar a un analista

delictivo, como enviar correos electrónicos automatizados, elaborar tablas del año hasta la fecha y crear gráficos de líneas. Pero no trato en detalle aspectos como la distribución de Poisson para analizar tasas de criminalidad o por qué el análisis de zonas calientes es importante.

Aunque el material es sin duda relevante para *todas las personas* que necesitan realizar análisis de datos usando python, espero utilizar ejemplos más realistas de análisis criminal para ilustrar mejor la utilidad de usar python para llevar a cabo análisis en tus tareas cotidianas como analista criminal.

¿Por qué aprender a programar?

Los analistas criminales realizan gran parte de su trabajo cuantitativo en hojas de cálculo (p. ej., Excel), y luego un número menor utiliza herramientas adicionales, como bases de datos (p. ej., Access, SQLServer), documentos formateados (Word, Powerpoint, PDF) y herramientas SIG (como ArcMap de ESRI). ¿Por qué molestarse en aprender python? Estoy de acuerdo en que Excel puede utilizarse para hacer cosas asombrosas con los datos, y muchas tareas son *intercambiables* entre python y una (o varias) herramientas diferentes.

Las ventajas de usar la programación, en lugar de herramientas que utilizan una interfaz gráfica de usuario (p. ej., señalar y hacer clic en la *GUI*), son:

- las tareas pueden automatizarse por completo
- las tareas están completamente documentadas

El primer punto de la lista es un argumento basado en el posible ahorro de tiempo al automatizar tareas. Supongamos que te toma 30 minutos realizar una tarea a diario. Si dedicas 100 horas a escribir código en Python para automatizar por completo la tarea, te habrás ahorrado tiempo en un plazo de 50 días gracias al proceso automatizado.

Sin embargo, para muchos informes periódicos en los que trabajan los analistas delictivos, este argumento de ahorro de tiempo puede no resultar convincente. Por ejemplo, cuando trabajaba como analista, tenía un informe mensual de CompStat con varios gráficos y mapas. Usando herramientas de interfaz gráfica (GUI), quizá me llevaba 24 horas (tres días laborables) completarlo. Una vez que escribí un código para crear automáticamente los gráficos, era una tarea de menos de un día. Pero quizá dediqué más de 160 horas a escribir el código para automatizar esa tarea. Se tardaría más de un año en alcanzar el punto de equilibrio en términos de ahorro de tiempo.

Muchos de los informes habituales que elaboran los analistas de criminalidad se parecerán a estos últimos; serán solo semirregulares, por lo que el argumento del ahorro de tiempo para automatizar mediante código no es tan contundente. (La automatización mediante código tiene más sentido, en términos de ahorro de tiempo, para tareas que deben realizarse con mayor frecuencia). Incluso en esos casos de informes semirregulares, sin embargo, creo que sigue valiendo la pena escribir código para automatizar todo lo posible.

Esto se debe al segundo punto – las tareas, cuando se escriben en código, por su naturaleza quedan completamente documentadas. Esto le permite a un analista, de forma retrospectiva, decir cosas como "este número se ve raro, ¿cómo lo calculé?", o que, cuando llega un nuevo analista y asume el trabajo, puedas decir "solo ve a revisar los scripts en la carpeta X". Contar con código estandarizado proporciona

un entorno mucho más profesional y transparente, lo cual es útil tanto para ti como analista como para la organización en su conjunto.

Además te permite escalar en tu trabajo. Si necesitas comprometer tu tiempo de forma indefinida para una tarea específica, incluso si es solo un día al mes para un informe concreto, solo podrás ampliar el alcance del trabajo que realizas hasta cierto punto. Poder automatizar las tareas tediosas es un paso necesario para liberar tiempo y dedicarlo a otros proyectos. Incluso te permite tomarte vacaciones, y los requisitos de informes pueden seguir cumpliéndose. En última instancia, aprender a programar probablemente te hará más productivo al realizar análisis de datos ad hoc, además de hacerte más competitivo en una gama más amplia de empleos (como los de ciencia de datos en el sector privado).

¿Por qué usar Python?

La sección anterior solo describe por qué alguien querría escribir código para automatizar tareas; no detalla por qué usar python específicamente (en lugar de, por ejemplo, R u otro programa estadístico). Además de python, he utilizado SPSS (un programa de pago) y R (otro programa estadístico de código abierto) de forma bastante extensa a lo largo de mi carrera. Tengo un paquete de R, ptools, para funciones comunes de interés para analistas del crimen, por ejemplo.

He migrado casi por completo mi trabajo de programación personal a python y ya no uso estas otras herramientas con mucha frecuencia. De nuevo, python es muy intercambiable con R para muchas tareas, pero en este punto de mi carrera prefiero python debido a su capacidad para gestionar proyectos completos, no solo realizar una única tarea. Además, muchos puestos de ciencia de datos en el sector privado se enfocan casi por completo en python (y menos en R). Así que creo que, en términos de desarrollo profesional, especialmente si tienes el objetivo de ampliar tus habilidades para optar a puestos de ciencia de datos en el sector privado, python es una mejor opción que R.

Hay situaciones en las que las herramientas de pago también son apropiadas. Programas estadísticos como SPSS y SAS no almacenan todo su conjunto de datos en la memoria, por lo que pueden ser muy prácticos para algunas tareas con grandes volúmenes de datos. Las herramientas GIS de ESRI (*Sistema de Información Geográfica*) pueden ser más convenientes para tareas de cartografía específicas (como calcular distancias de red o realizar geocodificación) que muchas de las soluciones de código abierto. (Además, las herramientas de ESRI pueden automatizarse también usando código de Python, por lo que no son mutuamente excluyentes.) Dicho esto, puedo aprovechar Python para casi el 100% de mis tareas cotidianas. Esto es especialmente importante para los analistas de criminalidad del sector público, ya que puede que no cuenten con presupuesto para adquirir programas de código cerrado. Python es 100% gratuito y de código abierto.

¿Cómo leer este libro?

Creo que la manera óptima de aprovechar el material de este libro es mediante un proceso de dos pasos. Tu nivel de experiencia con Python (ya sea algo de experiencia o ninguna) alterará en qué materiales te enfocas y cuáles probablemente puedes omitir. Para todos, sugeriría *ojear* brevemente cada capítulo desde el principio y comprender las metas de alto nivel que cada capítulo intenta enseñar.

Así es como yo personalmente consumo material técnico. Necesitas entender los objetivos de alto nivel que cualquier fragmento de código en particular pretende lograr antes de poder comprender los detalles técnicos más precisos. Si no puedes entender los objetivos de alto nivel, será muy difícil entender los detalles técnicos. También es útil comprender qué es posible hacer: no necesitas señalar y hacer clic en Excel para volver a generar ese informe de CompStat cada mes; puedes escribir código para automatizarlo (véase el Capítulo 10).

La segunda parte, después de la lectura rápida, depende de si eres neófito en Python o si tienes algo de experiencia previa en programación. Para aquellos neófitos sin experiencia, sugeriría que estudien en detalle los capítulos iniciales del 1 al 4 del libro. Un gran problema al aprender a ejecutar código es el problema de "empezar a ejecutar un ejemplo sencillo": descargar un programa y ejecutar comandos es un desafío para quienes nunca lo han hecho antes.

Esta parte – averiguar cómo instalar Python y ejecutar un comando sencillo puede ser el obstáculo más desafiante para empezar. Parte del desafío, como autor, es que los sistemas de cada persona son ligeramente diferentes y cambian con el tiempo. Las instrucciones para empezar tienden a ser muy particulares de tu computadora personal. Parte de la razón por la que estoy escribiendo este libro es que la mayoría de los materiales para principiantes ni siquiera intentan abordar este problema y usan trucos (como utilizar plataformas en línea) para ayudar a la gente a empezar.

Sin embargo, para realizar tareas reales que los analistas criminales necesitan para sus trabajos, no puedes usar las plataformas en línea. Muchas personas a las que se les enseña python en cursos universitarios utilizan dichas plataformas en línea (p. ej., si solo tienes experiencia usando notebooks de Jupyter o solo experiencia con notebooks de Google Collab), Necesitas saber cómo descargar python y ejecutarlo localmente en tu computadora personal para poder usarlo en tareas relacionadas con el trabajo. ¡Pero no te desesperes si estás teniendo problemas para empezar! Una técnica que utilizan los ingenieros de software profesionales se llama programación por pares (pair programming): busca a un amigo que sepa cómo ejecutar código de python (puedo ser yo, o alguien más de tu red), mírale por encima del hombro y luego pídele que te mire por encima del hombro. Esto te ayudará a empezar en el Capítulo 1.

Los capítulos 2 al 4 introducen objetos básicos (cadenas, números, listas, diccionarios) y acciones (sentencias condicionales, bucles, sustitución de cadenas). Estos son fundamentos de python muy aburridos — similar a cómo aprender la distribución normal resulta aburrido en tu clase introductoria de estadística, o aprender álgebra es aburrido en matemáticas. Sin embargo, son los bloques de construcción necesarios para entender cómo escribir código en python de manera eficaz. Quienes ya tengan experiencia más allá del nivel inicial pueden sentirse cómodos hojeando los capítulos 2-4; aun así, sugeriría examinarlos al menos de forma somera — probablemente se introduzcan algunas cosas que no sabías.

El capítulo 5 es una sección a la que, con frecuencia, ni siquiera quienes tienen experiencia introductoria están expuestos. Escribir tus propias funciones y entender cómo importarlas son un paso importante para pasar de escribir código como pasatiempo a crear un entorno profesional en el que desarrollar proyectos a lo largo del tiempo. De nuevo, muchas personas que han cursado en la universidad una asignatura de programación en Python no han estado expuestas a esto.

Los capítulos 6 al 9 se centran en mostrar ejemplos específicos de cómo trabajar y presentar datos que serán de amplio interés, no solo para quienes se dedican al análisis criminal, sino para cualquiera en un rol orientado a los datos. El capítulo 6 muestra las dos bibliotecas principales para trabajar con datos tabulares – `numpy` y `pandas`. Comprender la biblioteca pandas, en particular, es una habilidad importante para quienes usan Python para realizar análisis de datos.

El capítulo 7 muestra cómo usar Python para generar consultas SQL. Para quienes no están familiarizados con SQL, *Lenguaje de Consulta Estructurada*, SQL se utiliza para extraer datos de una base de datos externa y cargarlos en un dataframe de pandas. En este capítulo también presentaré diferentes sentencias SQL, ya que en algunos escenarios es mejor realizar ciertas tareas de análisis de datos en la base de datos *antes* de cargar los datos en un dataframe de pandas en memoria.

El Capítulo 8 presenta la biblioteca de gráficos `matplotlib` en Python. Crear gráficos de aspecto profesional es una habilidad importante para los analistas de datos. Generar gráficos de alta calidad es una señal para los destinatarios sobre la calidad del trabajo (para los analistas delictivos, estos podrían ser agentes de policía, el personal de mando o el público en general). Generar tales gráficos mediante código en Python es una buena manera de controlar el aspecto y la consistencia de los gráficos que produces.

El capítulo 9 introduce los cuadernos de Jupyter: ofrecen un entorno diferente al de la terminal para ejecutar código. Los cuadernos de Jupyter pueden combinar texto sin formato, celdas de código ejecutables y los resultados de esas ejecuciones (p. ej., gráficos y tablas). Este libro, internamente, se compila a partir de una serie de cuadernos de Jupyter. Presento Jupyter porque es una manera práctica de crear informes estandarizados que contienen distintos elementos del análisis de datos.

El capítulo final 10, *organización de proyectos*, aborda aspectos de la gestión de proyectos y la automatización del flujo de trabajo – los componentes finales necesarios para poder tomar proyectos simples y realmente aprovechar python para ayudarte a hacer tu trabajo como analista criminal. Ahora que sabes cómo escribir código, ¿cómo se ve un proyecto? Existen formas estándar en que deberías organizar tu proyecto, de modo que, ya sea que necesites volver a ejecutar el código o que otros lo necesiten, puedan entender los componentes necesarios. Esto implica cosas como crear un README que tenga información para replicar el entorno necesario para ejecutar el código, tener las funciones documentadas y almacenadas en una ubicación específica, y un punto de entrada claro que ejecute el código de forma automatizada.

El contenido general del libro pretende ir más allá de "cómo escribir código en Python", para ofrecer a las personas una experiencia integral de extremo a extremo al crear proyectos realistas que puedan ayudar a los analistas de delitos a realizar su trabajo. Esto implica algo más que ejecutar un único script, sino saber cómo hacen los profesionales cosas como consultar una base de datos, crear funciones reutilizables y configurar proyectos para automatizar diferentes tareas de análisis de datos a lo largo del tiempo.

Estas partes que no tratan de escribir código son lo que falta gravemente en los tutoriales actuales de programación en Python y constituyen la principal motivación para escribir el libro.

Mi trayectoria

Mientras cursaba mi doctorado en justicia penal en SUNY Albany (entre 2008 y 2015), trabajé en varios puestos de analista. Primero, en la División de Servicios de Justicia Penal del estado de Nueva York. Ese trabajo consistía principalmente en redactar informes estandarizados basados en la base de datos del historial de arrestos penales del estado de Nueva York. Luego, durante varios años trabajé internamente en el Departamento de Policía de Troy, NY, como su único analista criminal. Finalmente, trabajé como analista de investigación en el Finn Institute for Public Safety, una organización sin fines de lucro que colaboraba en proyectos de investigación con departamentos de policía del norte del estado de Nueva York.

Posteriormente fui profesor de criminología durante varios años en la Universidad de Texas en Dallas, desde 2016 hasta 2019. Durante ese tiempo escribí unas 40 publicaciones revisadas por pares y colaboré en proyectos cuantitativos con departamentos de policía de todo Estados Unidos. Presenté este trabajo con regularidad en la conferencia de la IACA y, durante un breve período, fui el presidente del comité de publicaciones de la IACA.

Actualmente (desde finales de 2019), he trabajado como científico de datos (en el sector privado) para una empresa de atención sanitaria. Mi trabajo ahora consiste en desarrollar un software centrado en el uso de modelos predictivos en relación con los datos de reclamaciones de atención médica. Aunque la atención sanitaria pueda parecer bastante diferente del análisis delictivo, muchos de los problemas son fundamentalmente los mismos (trabajar con reclamaciones de seguros de salud no es tan diferente de trabajar con reportes de delitos). Ejemplos de cosas que he desarrollado en mi empleo actual en el sector privado son modelos predictivos para identificar cuándo las reclamaciones se pagan de más o cuándo las personas presentan un alto riesgo de sufrir un infarto posterior.

Las habilidades necesarias para construir esos modelos predictivos no difieren del trabajo que he realizado al pronosticar la criminalidad en distintas áreas o al identificar a delincuentes crónicos con alto riesgo de cometer actos de violencia en el futuro. Por ello, mi experiencia personal como analista de criminalidad, investigador y luego científico de datos en el sector privado es lo que me motivó a escribir este libro. Quiero que parte de los avances más sofisticados en la academia, así como las prácticas de ingeniería de software del sector privado, se difundan con mayor amplitud en la profesión del análisis de criminalidad. Creo que un libro introductorio es el mejor método para lograr ese objetivo.

Comentarios sobre el libro

Para comentarios sobre el contenido del libro, puede enviarme un mensaje en https://crimede-coder.com/contact. No dude en enviarme comentarios, sugerir temas adicionales o avisarme de errores. Si está interesado en servicios más directos, como capacitación presencial para sus analistas o consultoría directa en proyectos en los que esté trabajando, también puede enviarme un correo electrónico. Entre los trabajos que he realizado para diversas agencias de justicia penal se encuentran la evaluación de programas, la redistritación, la automatización de distintos procesos, la consultoría en litigios civiles y la generación de modelos predictivos.

También tengo planes futuros para generar contenido de Python más avanzado. Estos incluyen libros sobre:

- programación más avanzada, datos y gestión de paquetes
- modelado de regresión
- aplicaciones de aprendizaje automático en el análisis delictivo
- análisis SIG con python

Los comentarios sobre el contenido y saber qué te interesa me ayudan a establecer prioridades para el trabajo futuro. Y que este libro en particular tenga más ventas también me motiva a escribir más – así que cuéntaselo a tus amigos si te gusta.

Agradecimientos

Agradezco a varios amigos por revisar los primeros borradores del libro y por proporcionar comentarios críticos. Renee Mitchell me dio comentarios tempranos sobre los borradores y fue uno de los principales impulsos para iniciar esta idea y llevarla a término. Dae-Young Kim ha sido diligente al darme comentarios muy detallados en cada capítulo, especialmente cuando necesito explicar el código con más detalle o cuando mi narrativa no es coherente con los ejemplos de código. Y agradezco a mi esposa por sus comentarios tempranos sobre los capítulos iniciales al escribir código en Python. (Con mucho, lo más difícil de programar es entender cómo ejecutar el código, no escribir el código en sí. Una buena prueba para ver si la documentación es suficiente es pedirle a alguien que *no* sea programador que vea si puede entenderla.)

Gracias a todos por el apoyo y a todos aquellos que compraron las versiones preliminares del libro.

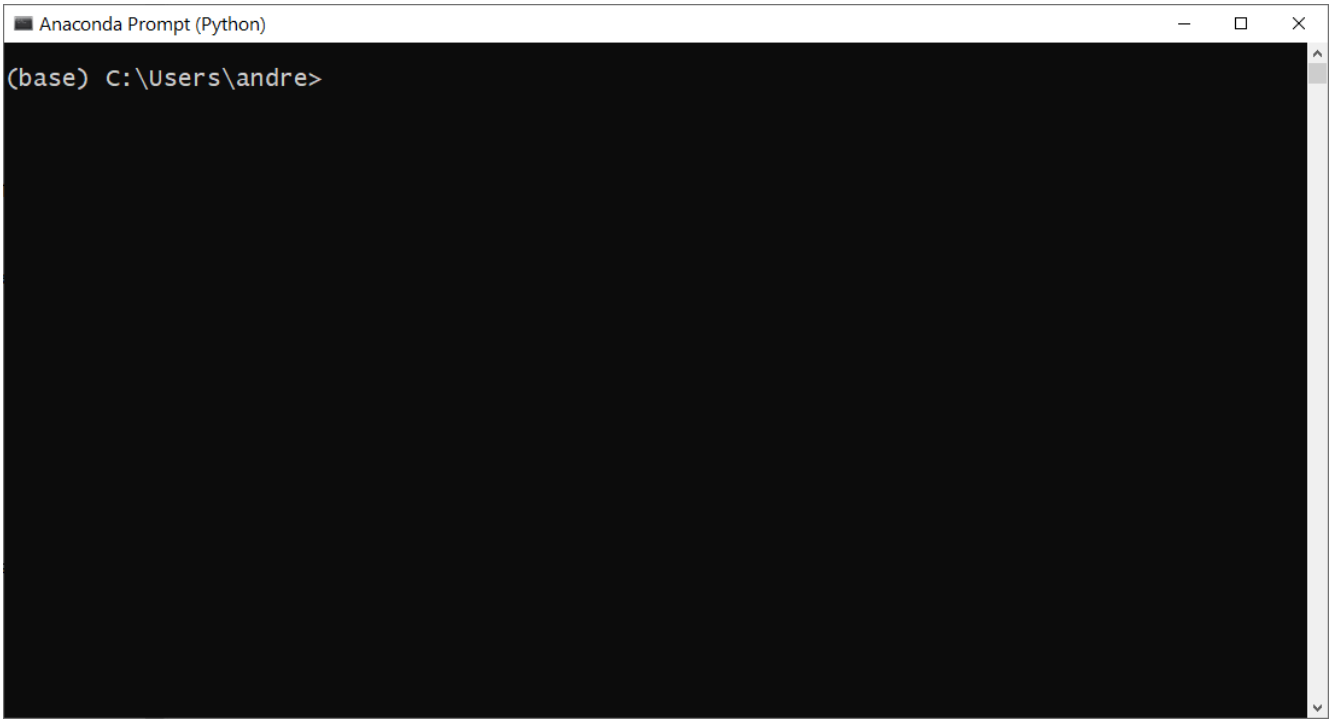
Este libro solo es posible gracias a diversas contribuciones de código abierto. Estoy usando el motor de Quarto para generar este libro en diferentes formatos (PDF y EPUB), que a su vez utiliza LaTeX y Pandoc bajo el capó. Python en sí es de código abierto, y uso ampliamente herramientas de Anaconda. Mi agradecimiento a todas aquellas personas que ayudan a que el mundo siga girando entre bastidores.

1 Configuración de Python

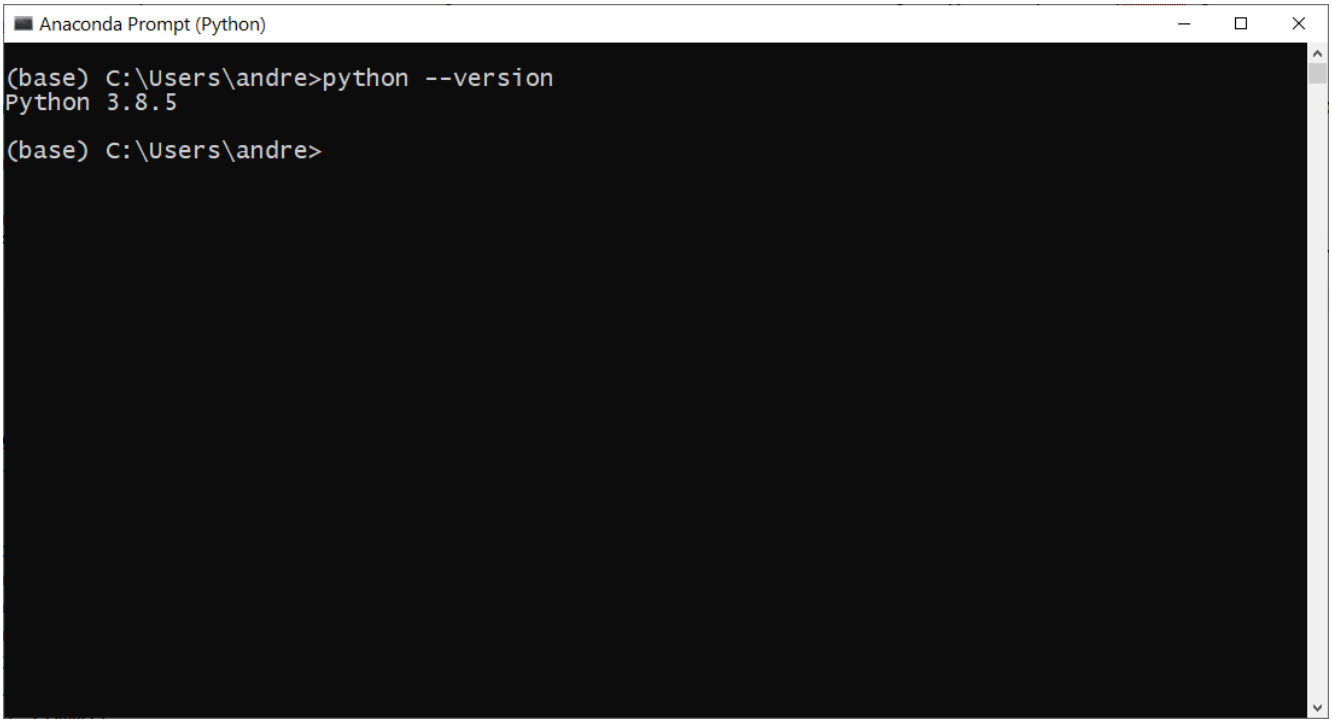
Primero, antes de poder empezar a *escribir un código*, necesitamos configurar python en tu equipo local. Mi recomendación para los analistas del crimen es instalar la versión de python de Anaconda, que se puede descargar para tu equipo en https://www.anaconda.com/download. Estoy escribiendo este libro en Windows, pero la mayoría de mis consejos también deberían aplicarse a usuarios de Mac y Unix (puedes descargar Anaconda y ejecutar python en cualquiera de esos sistemas operativos). Cuando pueda haber alguna diferencia, proporcionaré una nota destacada. Por ejemplo:

> **Note**
>
> Al usar diferentes rutas a ubicaciones de archivos, las máquinas Windows usan barras invertidas, p. ej., `C:\Users\andre`, mientras que las máquinas Mac y Unix usan barras diagonales, `/users/andre`.

Una vez que Anaconda esté instalado, abre el *símbolo del sistema de Anaconda* (trátalo igual que cualquier programa que tengas instalado en tu equipo; en Windows puedes navegar por los programas que aparecen al hacer clic en el icono de Windows en la esquina inferior izquierda). Una vez que el símbolo del sistema de Anaconda esté abierto, debería verse parecido a lo siguiente.

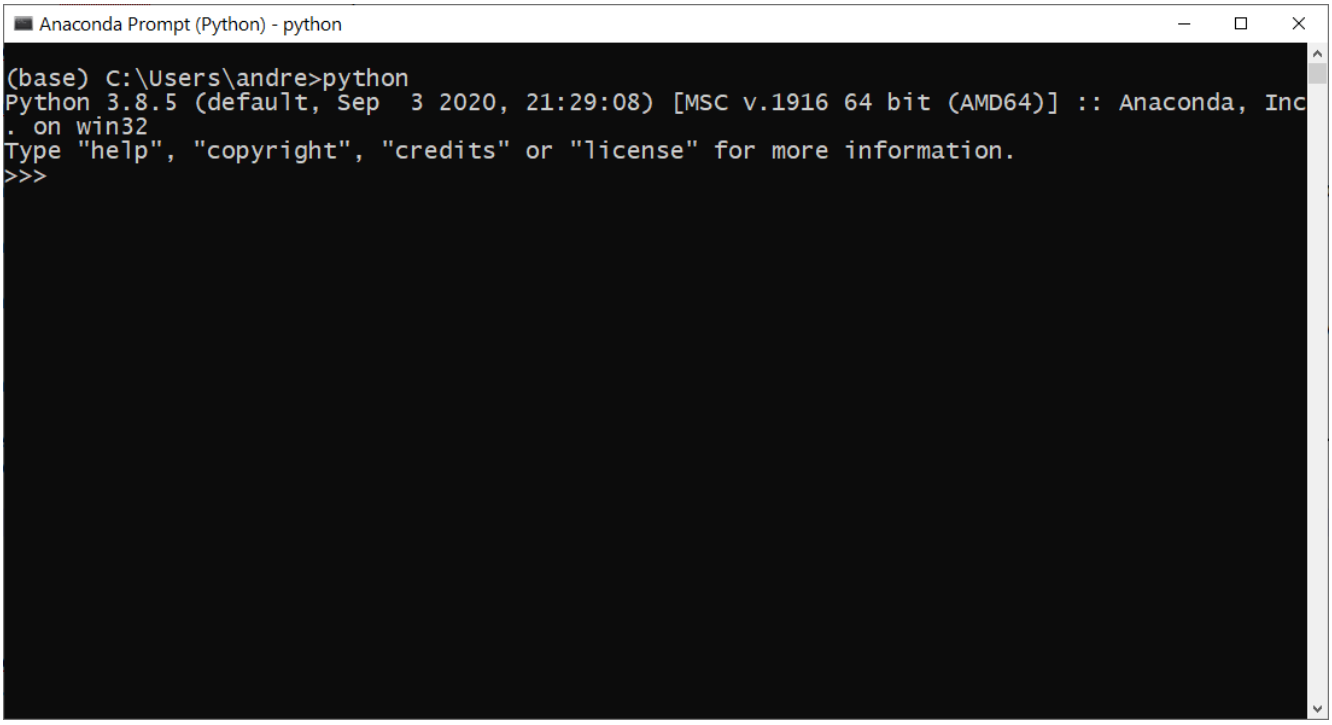


Adelante, escribe `python --version` en la línea de comandos, presiona Enter y observa el resultado. Esto te dirá si has instalado Python correctamente. Mi versión de Python es `3.8.5`. Tu versión puede ser diferente de la que utilicé para escribir este libro, pero para el contenido que abordaré en este libro no supondrá ninguna diferencia.

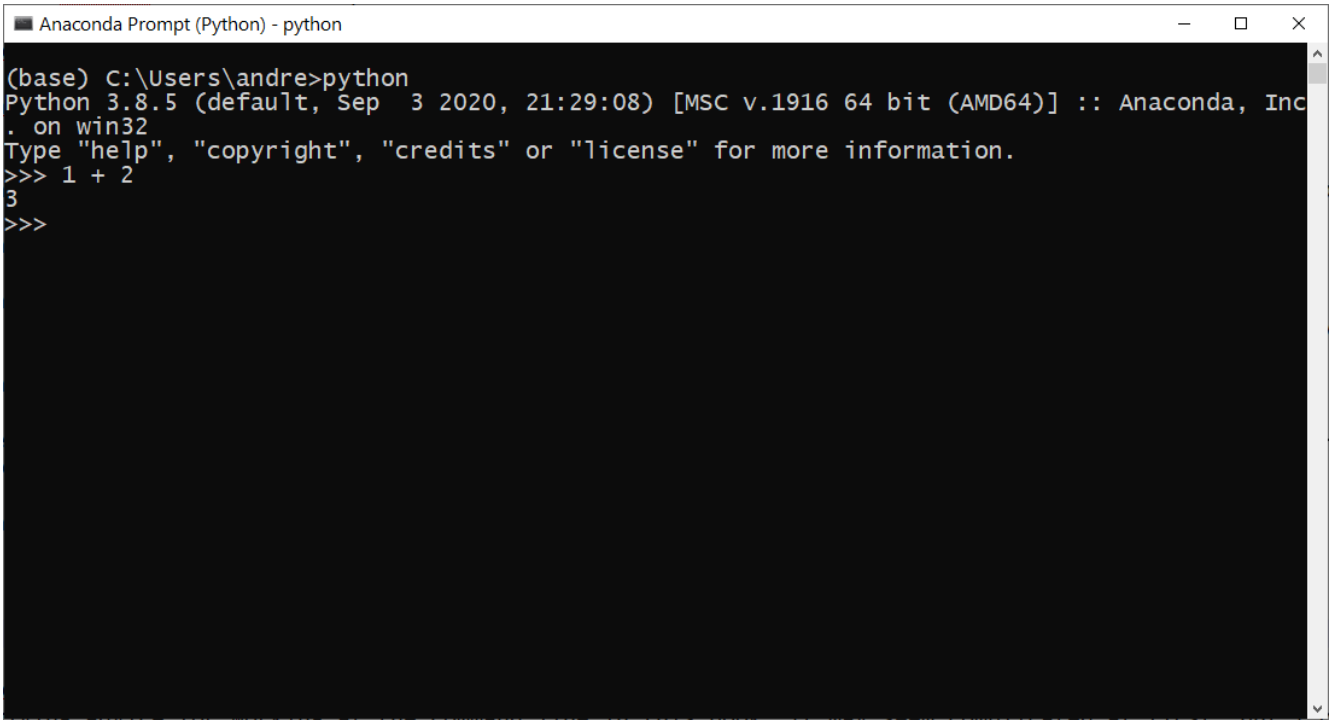


1.1 Ejecución en el REPL

Ahora vamos a ejecutar una sesión interactiva de Python; a veces la gente la llama el *REPL*, bucle-leer-evaluar-imprimir (read-eval-print-loop en inglés). Simplemente escribe `python` en el símbolo del sistema y presiona Enter. Entonces verás esta pantalla y estarás dentro de una sesión de Python.



El cursor en la terminal debería estar ubicado en la parte `>>>` de la pantalla. Ahora, en el prompt, simplemente escribe `1 + 2`, presiona Enter y mostrará la respuesta:



Felicidades, ahora has escrito un código en Python.

Note

Observe que la ubicación del cursor en la terminal en este punto debería estar en la línea `>>>`. No puede subir a una línea anterior en la terminal, como puede hacerlo en un editor de texto, pero sí puede editar elementos en una sola línea en la terminal. Así que puede escribir `1 + 2` y, antes de pulsar Enter, presione la tecla de retroceso y edite la línea para que sea `1 + 3`.

1.2 Ejecutar un script de Python

Ahora hagamos un script de Python sencillo y llamemos a ese script desde la línea de comandos. Primero, navega a cualquier carpeta de tu equipo en la que puedas añadir un archivo (consulta la nota de abajo para navegar a diferentes ubicaciones mediante la línea de comandos). Aquí, navegué a la carpeta en mi equipo `B:\code_examples`, pero la ubicación de tu carpeta probablemente será diferente. Crea un archivo simple, llámalo `hello.py`, en esa misma carpeta (puedes inicializar el archivo simplemente escribiendo `echo "" > hello.py` en el símbolo del sistema, o `touch hello.py` que es más fácil si estás en una máquina Mac/Unix. O en el sistema operativo Windows crea un archivo de texto y luego cambia la extensión a `.py` en lugar de `.txt`).

Note

Daré consejos para trabajar en la línea de comandos en este libro. Puede parecer complicado al principio, pero solo tengo memorizados unos pocos comandos. Para la gestión de proyectos, es importante saber exactamente *dónde* ejecutas los comandos. Aquí hay algunas notas sobre el uso de `cd` para navegar a diferentes directorios:

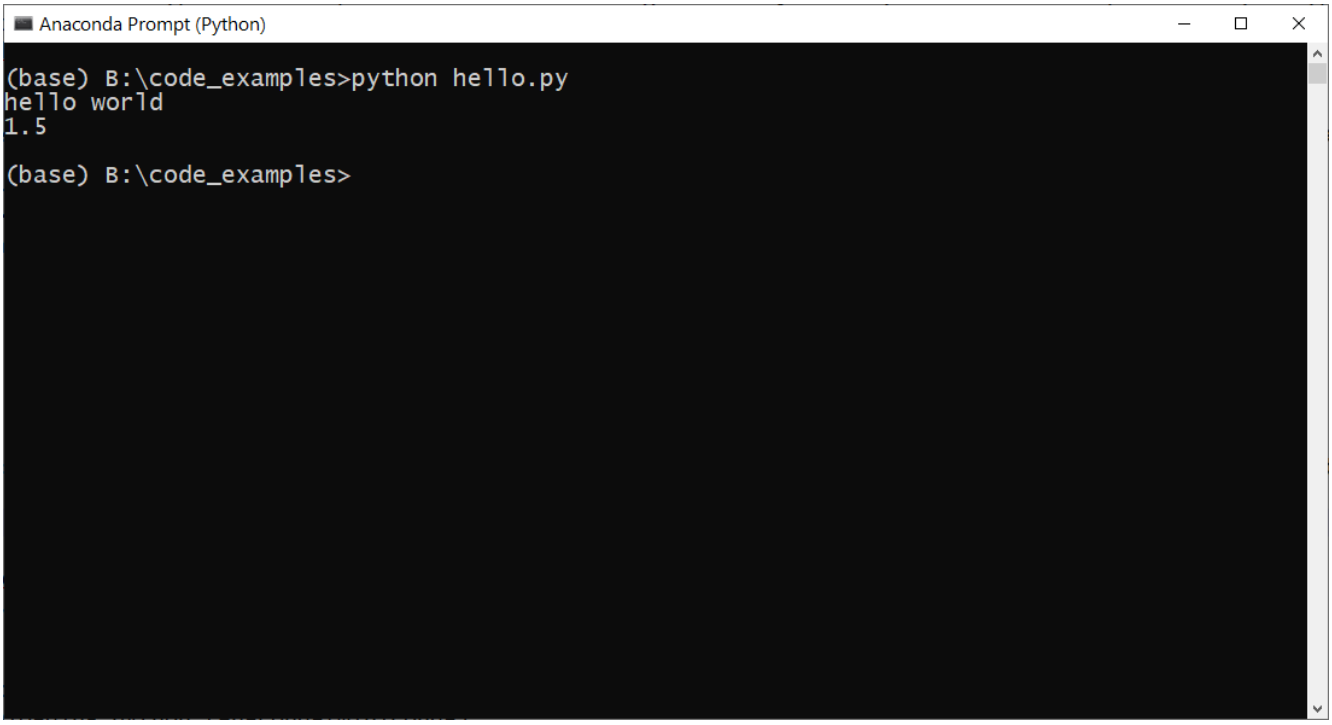
- usa `cd YourPath\YourFolder` para moverte a diferentes carpetas, p. ej., en Windows podría ser `cd D:\Dropbox\Project`, mientras que en Unix podría ser `cd /Project/sub_folder`. En Windows, para cambiar a una unidad diferente, puedes simplemente escribir esa letra de unidad (sin `cd` al principio; p. ej., si solo escribes `D:`, el símbolo del sistema cambiará el directorio a la unidad `D:`).
- Si tu ruta tiene un espacio, debe ir entre comillas al usar `cd`, por ejemplo `cd "C:\OneDrive\OneDrive - Uni\Folder"`. Sin las comillas, el comando `cd` dará un error.
- usa `cd ..\` (o `cd ../` en Unix/Mac) para subir una carpeta; p. ej., si estás en `D:\Project\sub_project` y escribes `cd ../`, ahora estarás en `D:\Project`.
- No necesitas escribir la ruta completa para bajar a una sola carpeta, así que si estás en `D:\Project` y escribes `cd .\sub_project` bajarás al directorio `D:\Project\sub_project`.
- usa el comando `pwd` para mostrar el directorio actual.

En ese archivo `hello.py` (que es solo un archivo de texto), ábrelo con el editor de texto que prefieras. Luego escribe estas líneas de código en ese archivo y guarda el archivo:

```
# This is a comment line
x = 'hello world'
print(x)

y = 3/2
print(y)
```

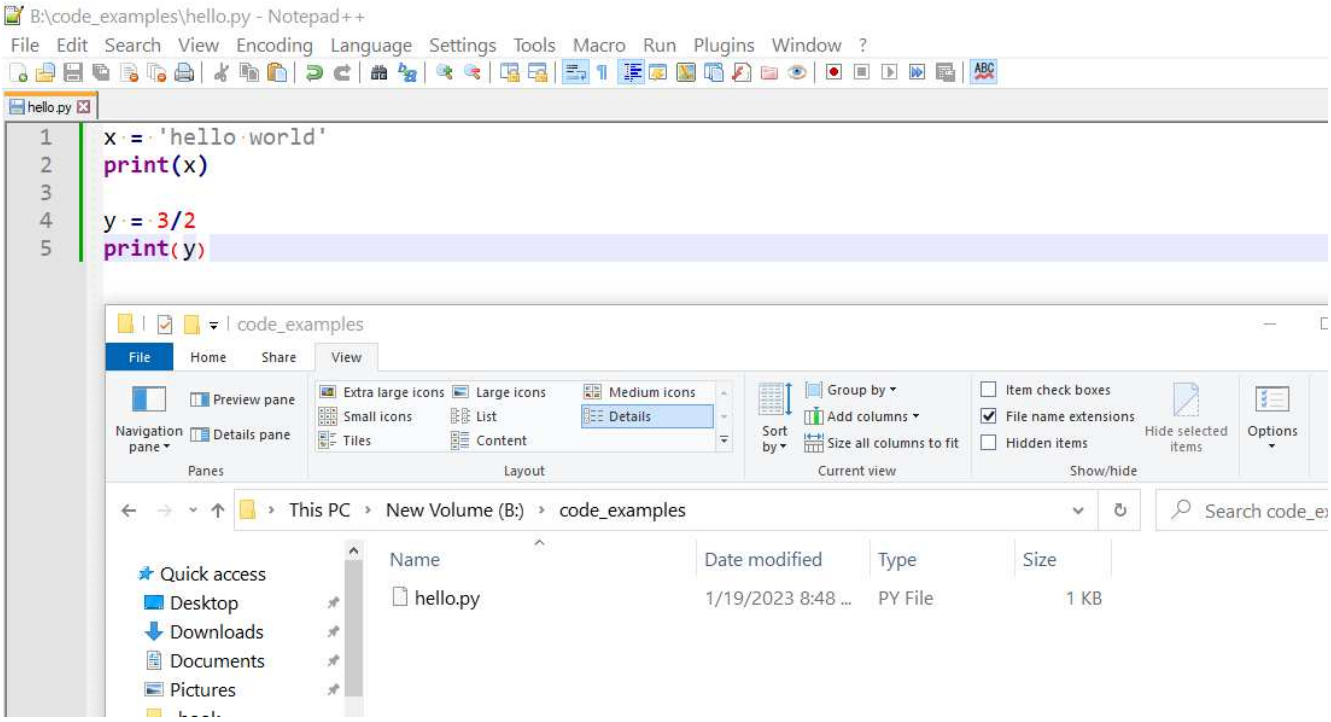
Ahora, de vuelta en el símbolo del sistema, escribe `python hello.py` y presiona Enter. Deberías ver que se ejecutó tu archivo de Python y que se imprimieron los resultados.



Ciertos escenarios determinarán si estás escribiendo código en una sesión interactiva de REPL o ejecutando código mediante scripts. En la mayor parte de mi trabajo, depuro el código inicial usando el REPL y luego guardo el código final y ejecuto todo el conjunto de procedimientos mediante un script.

1.3 Algunas notas adicionales

Uso Notepad++ para escribir gran parte de mi código. Aquí se muestra cómo se ve el bloque de código anterior, con Notepad++ y el propio archivo:



Notepad++ sabe que un archivo `.py` es un archivo de Python, y por lo tanto ofrece un buen formato de código. También permite configurar una opción para ver los espacios en blanco, lo cual será más importante al escribir bloques de código condicional en Python, lo que se verá más adelante. (No deberías escribir código en un programa que formatea tu texto, como Microsoft Word, ya que su autoformato puede causar errores en tu código.)

Pero hay otras opciones que pueden ayudarte a escribir código en Python. En ocasiones, en mi trabajo utilizo Visual Studio (VS) Code, que es todo un IDE (entorno de desarrollo integrado). Esto solo significa que tiene cosas adicionales (como una consola de comandos integrada y soporte para GitHub). VS Code tiene extensiones para Python, así como para varios otros lenguajes. Otro IDE popular para Python es PyCharm. Estos, nuevamente, son en su mayoría intercambiables; a mí me gusta Notepad++ por su sencillez.

> **Note**
>
> Aquí hay algunos comandos adicionales de la línea de comandos que uso habitualmente:
>
> - En Windows, para limpiar la terminal puedes usar `cls`; en Unix/Mac puedes usar `clear`
> - puedes usar `cat file.py` para mostrar en la terminal el contenido de un archivo
> - puedes usar `mkdir` para crear una nueva carpeta
> - puedes redirigir la salida a un archivo; p. ej., `python hello.py > log.txt` guardaría la salida del comando anterior en el archivo `log.txt` en lugar de mostrarla en la terminal.
> - si usas `python script.py >> log.txt`, esto *anexa* la salida al archivo de registro. Lo cual es útil para tareas repetitivas que se actualizan con el tiempo.

Otra opción para escribir código de Python (especialmente para quienes realizan computación científica)

es usar Jupyter Notebooks. Muchos tutoriales de Python para principiantes sugieren esto. Más adelante incluiré un capítulo que muestra cómo crear un informe estandarizado con Jupyter Notebooks, pero no recomiendo esta herramienta para quienes recién comienzan. Muchos de los aspectos sobre gestión de proyectos que trataré en este libro son difíciles de controlar con Jupyter.

Ahora que tienes los conocimientos para ejecutar código de Python, el próximo capítulo cubrirá más conceptos básicos sobre cómo escribir código de Python.

2 Primeros pasos para escribir un código en Python

Aunque el capítulo anterior te mostró cómo ejecutar un código, ya sea en el REPL o mediante scripts, este capítulo te pondrá en marcha escribiendo un código de Python propiamente dicho. Recomiendo teclear los ejemplos de código que aparecen en los recuadros en una sesión de REPL para seguirlos, aunque también puedes guardarlos en scripts `.py` y ejecutarlos directamente. Las partes en gris son lo que escribirías en la terminal, y los recuadros azules son lo que debería imprimirse.

Este capítulo presentará una introducción al trabajo con números, cadenas, booleanos, listas y diccionarios. Estos *objetos* son los componentes básicos de prácticamente todo el código en Python.

> Note
>
> En una primera lectura, este capítulo (quizá todos) parecerá mucha información y probablemente te resultará aburrido. Sugiero que lo sigas en el REPL, pero que *los leas por encima* con bastante rapidez (especialmente los capítulos 2, 3 y 4 sobre los conceptos básicos de objetos, cadenas y bucles). He intentado intencionalmente ser bastante exhaustivo con lo básico. Al trabajar en proyectos reales, puede que necesites volver a los capítulos para entender y volver a familiarizarte con los temas. No te conviertes en experto por leer un capítulo una sola vez, sino por escribir código repetidamente para tus propios proyectos con el tiempo.

2.1 Valores numéricos

Para empezar, puedes pensar en el código de Python como *objetos* y *operaciones* aplicadas a esos objetos. Por ejemplo, si ejecutas el código de Python:

```
x = 1
y = x + 1
print(y)
```

```
2
```

Aquí primero creamos un objeto, `x` que es *asignado* un valor de 1 mediante el símbolo `=`. Luego creamos un segundo objeto, `y`, que es asignado el valor `x + 1`. Finalmente, hacemos `print` del valor del objeto `y`. `print` es una función cuya única finalidad es mostrar el valor (o, más específicamente, la representación

en forma de cadena de ese objeto) en la terminal (o en cualquier ubicación a la que le indiques a python que envíe sus resultados).

> Note
>
> En el REPL, también es posible escribir un solo elemento en una línea y se imprimirá la salida en la terminal. Así que, en lugar de `print(y)`, en una sesión de REPL podrías simplemente escribir `y` y obtendrás el mismo resultado. En un script, sin embargo, solo `print(y)` envía la salida a la terminal.

En el ejemplo anterior, los objetos eran valores enteros. También puedes tener objetos números decimales flotante. Python, a diferencia de algunos lenguajes de programación, no te obliga a definir de antemano el tipo de objeto.

```
x = -1
y = 3.2
z = x/y
print(z)
```

```
-0.3125
```

Cuando se trabaja con valores numéricos, Python es inteligente y convierte `z` aquí en un valor de tipo float, aunque use un entero como entrada (incluso con dos enteros, p. ej., `z = 1/2`, en Python `z` será un float). Puedes ver que aquí hice una división; la mayoría de las operaciones matemáticas son similares a lo que escribirías en cualquier calculadora, con la excepción de que las potencias usan `**` en lugar de `^`:

```
# Showing off the different
# math operations
x = 2
print(x - 1)
print(x + 1)
print(x/2)
print(x*2)
print(x**3)      # x to the third power
print(x**(1/2))  # x to the 1/2 power (square root)
```

```
1
3
1.0
4
8
1.4142135623730951
```

Uno de los operadores especiales en Python sirve para modificar el valor numérico de un objeto. Por ejemplo, para incrementar el valor de un objeto en uno, podrías hacer:

```
x = 1
x = x + 1
print(x)
```

```
2
```

Pero es más fácil usar la notación especial de `x += 1`:

```
x = 1
x += 1
print(x)
```

```
2
```

Ten en cuenta que también puedes realizar otras operaciones matemáticas, como restar `x -= 1`, multiplicar `x *= 2`, dividir `x /= 2` o elevar a una potencia `x **= 2`.

Los últimos ejemplos numéricos básicos que se deben presentar son `//`, la división entera, y `%`, el operador de módulo. La división entera solo devuelve los números enteros en un problema de división, y el módulo es el resto del problema de división.

```
x = 5
print(x//2)
print(x % 2)
```

```
2
1
```

En capítulos posteriores en los que se analizan bibliotecas destinadas a trabajar con datos tabulares (numpy y pandas), abordaré el procesamiento de datos numéricos con mayor detalle. Como a menudo no se desea hacer cálculos sobre un solo objeto, sino sobre un vector de múltiples objetos.

2.2 Cadenas

Otro objeto básico que usarás a menudo en Python son las cadenas de caracteres. Si encierras un conjunto de caracteres entre comillas, eso da como resultado un objeto cadena. Aquí incluso realizo la suma de objetos cadena, lo que concatena las dos cadenas.

```
x = "ABC"
y = "de"
z = x + y
print(z)
```

```
ABCde
```

Las cadenas pueden contener cualquier carácter alfanumérico, por lo que pueden albergar representaciones textuales de números. Ten en cuenta que sumar dos cadenas no es lo mismo que sumar dos valores numéricos: ¡concatena las dos cadenas!

```
x = "1"
y = "3"
z = x + y
print(z)
```

```
13
```

Puedes convertir cadenas a valores numéricos mediante las funciones `int` y `float`.

```
x = "1"
xi = int(x)
xf = float(x)
print(xi,",",xf) # can print multiple objects at once
```

```
1 , 1.0
```

Y, viceversa, puedes convertir valores numéricos en cadenas mediante la función `str`.

```
xi = 1
xf = 1.0 # if you use decimal, will be float

xsi = str(xi)
xsf = str(xf)

print(xsi + "|" + xsf) #concat the strings
```

```
1|1.0
```

Note

He presentado tres funciones hasta ahora, `print`, `int`, `float` y `str`. Las funciones en python adoptan la forma `function(input)`, es decir, un nombre particular seguido de dos paréntesis.

Las cadenas pueden encerrarse con comillas simples o dobles. Sin embargo, ten en cuenta que los caracteres especiales en las cadenas pueden causar problemas. Las barras invertidas en las variables de ruta de Windows son un ejemplo común:

```
project_path = "C:\Project\SubFolder"
project_path # Note no print statement
```

```
'C:\\Project\\SubFolder'
```

Note

Lo que se imprime en la consola no es necesariamente la misma representación textual del objeto en sí. Por ejemplo, prueba `x = "Line1\nLine2"` y luego escribe solo `x` en la terminal y observa qué se imprime, a diferencia de escribir `print(x)`. En este ejemplo, print interpreta los saltos de línea en la cadena, mientras que simplemente escribir `x` en una sesión REPL no lo hace.

¡Puedes ver que python insertó barras invertidas adicionales! Si quieres que la cadena quede exactamente como la escribiste, puedes usar la opción de cadena `r""`, que significa cadena *real*.

```
project_path = r"C:\NoExtra\BackSlashes"
print(project_path)
```

```
C:\NoExtra\BackSlashes
```

Si quieres una cadena de varias líneas en Python, puedes usar comillas triples. Ten en cuenta que esta cadena tiene saltos de línea en el objeto de cadena.

```
long_note = '''
This is a long
string note
over multiple lines
'''

print(long_note)
```

```
This is a long
string note
over multiple lines
```

Si tienes una cadena muy larga, como una URL, en la que no quieres insertar saltos de línea, puedes envolver la cadena entre paréntesis. El intérprete de Python, al final, simplemente lo convierte en una cadena:

```
long_url = ("Pretend I am a super "
            "long url on a single line")

print(long_url)
```

```
Pretend I am a super long url on a single line
```

Tengo todo un capítulo, el Capítulo 3, dedicado al uso más avanzado de las cadenas de texto.

2.3 Booleanos

Los objetos booleanos solo pueden tener dos valores, `True` o `False`.

```
print(1 == 1)
print(2 == 3)
```

```
True
False
```

Puedes ver que uso `==` para hacer una comparación de igualdad. Recuerda que un solo `=` es para asignación. Para indicar que no es igual, el símbolo es `!=`:

```
print('a' != 'a')
print('b' != 'c')
```

```
False
True
```

Y luego también se puede usar la lógica de menor que y mayor que:

```
print(1 < 1)
print(1 <= 1)
print(2 > 1)
print(2 >= 1)
```

```
False
True
True
True
```

Nota: también se pueden hacer comparaciones (de menor que) con cadenas; por ejemplo, `'a' < 'b'` es `True`, pero no utilizo esto muy frecuentemente. El ejemplo de abajo muestra un caso que probablemente no era el pretendido, ya que accidentalmente compara representaciones en cadena de números en lugar de números directamente.

```
print('10' < '2')
```

```
True
```

A menudo se utiliza un booleano para realizar lógica condicional en código de Python. Así puedes tener una instrucción `if` como se muestra a continuación:

```
a = 1

if a == 1:
  print(a + 1)
```

```
2
```

Una característica particular de la sintaxis de python es que el espacio en blanco es significativo. Para indicar que la línea `print` está dentro de la sentencia if, anteponemos varios espacios al principio. El número de espacios no importa, aunque debe ser consistente. (Y técnicamente también puedes usar tabulaciones en lugar de espacios, pero lo desaconsejo encarecidamente, ya que puede hacer que editar los archivos sea un fastidio.)

Python usa `if`, `elif`, `else` para las sentencias condicionales. Aquí hay un ejemplo con if y else:

```
a = 1

if a != 1:
  print('a does not equal 1')
```

```
  print(a + 1)
else:
  print('I am in the else part')
```

```
I am in the else part
```

La forma en que funcionan estas instrucciones es que se verifica si la primera instrucción `if` es verdadera. Si esa instrucción es verdadera, ejecuta el código anidado en la parte del if, *y luego sale del bloque de código*. Si la instrucción `if` se evalúa como False, entonces ejecuta el bloque de la instrucción `else` para todos los demás casos.

A veces querrás encadenar varias comprobaciones, p. ej., "si A, haz Y; de lo contrario, si B, haz X". Para hacer eso en código de Python, usarías `if` y luego `elif`.

```
a = 1

if a != 1:
  print('a does not equal 1')
  print(a + 1)
elif a == 1:
  print('I am in the elif part')
else:
  print('I am in the else part')
```

```
I am in the elif part
```

Si un `elif` es verdadero, ejecuta ese bloque y luego sale, igual que la instrucción `if`. Así que, en el fragmento de código anterior, como la instrucción `elif` es verdadera, nunca llega a la parte `else` de la lógica condicional.

No hay nada que conecte las distintas sentencias condicionales entre sí, por lo que no es necesario que todas las secciones hagan referencia al mismo elemento:

```
a = 1
b = 'X'

if a != 1:
  print('a does not equal 1')
elif a == 2:
  print('a equals 2')
elif b == 'X':
  print('I am in the b check elif part')
else:
```

```
  print('I am in the else part')
```

```
I am in the b check elif part
```

Y también puedes escribir lógica condicional adicional dentro de una estructura condicional.

```
a = 1
b = 'X'

if a != 1:
  print('a does not equal 1')
  if b == 'X':
    print('b check in first if')
else:
  print('I am in the else part')
  if b == 'X':
    print('b check in elif')
```

```
I am in the else part
b check in elif
```

A veces, en una parte de la condición, no quieres hacer nada. En esos casos, puedes usar `pass` dentro de la condición.

```
a = 1

if a == 1:
  pass # This snippet will print nothing
else:
  print('I am in the else part')
```

Normalmente se desea poner primero las condiciones más comunes en una serie de sentencias if, y dejar las menos comunes más abajo. Así, si en la condición más común no haces nada y solo en condiciones más raras realizas alguna acción, esta es una manera perfectamente razonable de escribir tus sentencias if.

Estos ejemplos comparan solo dos objetos, pero puedes combinar múltiples sentencias condicionales usando `and` y `or`.

```
# and example
a = (1 == 1) and (2 == 2)
print(a)
```

```
b = (1 == 1) and (2 == 3)
print(b)

# or example
c = (1 == 1) or (2 == 3)
print(c)
```

```
True
False
True
```

Sin embargo, hay operadores abreviados: la y comercial para `and` y la barra vertical para `or`:

```
# ampersand for and
a = (1 == 1) & (2 == 2)
print(a)

b = (1 == 1) & (2 == 3)
print(b)

# pipe for or
c = (1 == 1) | (2 == 3)
print(c)
```

```
True
False
True
```

Técnicamente no necesitas los paréntesis en los ejemplos anteriores, pero a mí me resulta mucho más fácil leer el código así y mantener juntos distintos términos:

```
#       This is false        But this is true
a = ((1 == 2) & (2 == 2)) | (4 == 4)
print(a)
```

```
True
```

Pero la mayor parte del tiempo, si es posible, simplemente lo descompongo en código y hago que la línea final de la instrucción if sea lo más simple posible.

```
check1 = (1 == 2) & (2 == 2) # This is false
check2 = (4 == 4)            # This is true

if check1 & check2:
  print('Both check1 and check2 are true')
elif check1 | check2:
  print('At least one of check1 or check2 are true')
else:
  print('Neither check1 or check2 are true')
```

```
At least one of check1 or check2 are true
```

En ocasiones no se utilizan los operadores matemáticos para generar las expresiones booleanas, sino `is` o `is not`. Quizás el uso más común de esto es con el objeto `None`, que puede considerarse un valor faltante en Python.

```
x = None

if x is None:
  print('x has no value')
else:
  print('x has some value')
```

```
x has no value
```

Técnicamente, cuando escribes `a is b`, no solo verifica los valores de los objetos, sino también que haga referencia al *mismo objeto exacto* en memoria. Técnicamente, `x == None` funcionará en el ejemplo anterior, pero es práctica común escribirlo de la forma `x is None`. También puedes comprobar la condición opuesta mediante `is not None`:

```
x = None

if x is not None:
  print('x has some value')
else:
  print('x is None')
```

```
x is None
```

Como ejemplo final, se pueden omitir por completo los operadores de comparación o las sentencias (if). Si pasas un objeto "vacío" a una sentencia if, Python comprueba si el objeto tiene algún elemento y devuelve `True` si es así. Así que si pasas una cadena vacía, la sentencia if devuelve `False` aquí:

```
x = ''

if x:
  print('x has some value')
else:
  print('x is an empty string')
```

```
x is an empty string
```

Esto funciona para otros objetos de Python, como listas y diccionarios vacíos, lo cual se ilustrará en la sección siguiente.

2.4 Listas

Es importante para los principiantes comprender los diferentes objetos que son contenedores de otros objetos. El primero es un objeto de tipo *lista*. Una lista contiene varios otros objetos, y se crea una lista colocando elementos dentro de corchetes y separando los elementos con comas. ¡Es más fácil mostrarlo que explicarlo con palabras!

```
x = [1,2,3]
print(x)
```

```
[1, 2, 3]
```

Las listas pueden contener distintos tipos de objetos; por ejemplo, pueden incluir tanto cadenas como valores numéricos en la misma lista. Aquí también muestro que las listas pueden contener otros objetos de Python definidos previamente.

```
a = 1
b = 'z'
x = [a,b,3]
print(x)
```

```
[1, 'z', 3]
```

Puedes dividir los elementos de una lista en varias líneas en el intérprete de Python; detecta el corchete de cierre para saber que la entrada de la lista ha terminado. Así que la lista resultante a continuación es exactamente la misma que `x = [1,2,3]`, solo que es otra forma de escribirla (puede ser útil dividir listas muy largas en varias líneas para mejorar la legibilidad de tu código).

```
# it is ok to define lists
# over multiple lines
x = [1,
     2,
     3]

# it is the same as earlier
# on a single line
print(x)
```

```
[1, 2, 3]
```

Un error común al trabajar con listas es olvidar la coma. Con valores numéricos, esto a menudo dará como resultado un error, pero con cadenas a veces puede no dar lugar a un error, ya que Python simplemente concatenará implícitamente las cadenas. Por ejemplo:

```
# This is probably not what was intended
x = ['a' 'b', 'c']
print(x)
```

```
['ab', 'c']
```

Entonces, aquí se omitió la coma entre las dos primeras cadenas, por lo que la lista resultante tiene solo dos elementos, y el primero es `'ab'`.

> **Note**
>
> Una ventaja de escribir listas en varias líneas es que puedes usar edición en *columna* en varios editores de texto. En Notepad++, puedes mantener presionada la tecla Alt y seleccionar varias líneas para editarlas a la vez. La mayoría de los editores de texto avanzados tienen una función similar.

Puedes acceder a elementos individuales de una lista mediante su índice. Ten en cuenta que en Python los índices de las listas comienzan en 0, no en 1, por lo que el primer elemento de una lista tiene índice 0, el segundo elemento tiene índice 1, etc.

```
y = ['a','b','c']
print(y[0])
print(y[1])
print(y[2])
```

```
a
b
c
```

También puedes usar índices *negativos* para acceder a los elementos en orden inverso en una lista. Así, -1 accede al último elemento de una lista, -2 al penúltimo, etc.

```
y = ['a','b','c']
print(y[-1])
print(y[-2])
```

```
c
b
```

Por último, en cuanto al acceso a varios elementos de una lista, puedes usar la notación de corte (slice). Así, `x[1:3]` accedería al segundo y tercer elemento de la lista (es equivalente a la notación `[,)` en matemáticas, por lo que el extremo izquierdo del corte es cerrado y el extremo derecho es abierto).

```
y = ['a','b','c']
print(y[1:3]) # note it returns a list!
```

```
['b', 'c']
```

También puedes usar `x[1:]` para indicar, "dame el segundo elemento de la lista hasta el final", o `x[:3]` para "dame los primeros 3 elementos de la lista.

Se puede crear una lista vacía simplemente asignando `[]` a un valor. Otro truco útil que conviene conocer es que se puede hacer una comprobación booleana para una lista vacía.

```
x = []

if x:
  print('The x list is not empty')
else:
  print('x is empty')
```

```
x is empty
```

Una lista vacía no tiene elementos, así que si intentas acceder a uno generará un error. Como se mostró arriba, si intentas usar `x[0]`, obtendrás el error `index out of range`.

Otra operación booleana sobre listas consiste en comprobar si un elemento está contenido en esa lista.

```
x = ['a','b','c']

if 'd' in x:
  print('The list has a d element')
elif 'c' in x:
  print('The list has a c element')
else:
  print('x has neither d or c')
```

```
The list has a c element
```

Un dato importante que debes conocer sobre las listas es que son *mutables*. ¿Qué significa eso exactamente? Significa que podemos modificar el contenido de una lista. Así, por ejemplo, podemos reemplazar un único elemento en una lista.

```
y = ['a','b','c']
print(y)

y[1] = 'Z'
print(y)
```

```
['a', 'b', 'c']
['a', 'Z', 'c']
```

Para un ejemplo más complicado, las listas pueden apuntar a otras listas. Como las listas son mutables, puedes modificar la lista interna aquí y la lista externa refleja este cambio.

```
x = ['a','b','c']
y = [1, x]
print(y)

# if we alter x
# y points to
# the altered x list
x[1] = 'Z'
print(y)
```

```
[1, ['a', 'b', 'c']]
[1, ['a', 'Z', 'c']]
```

La forma de pensar en esto es que la lista `y` aquí no contiene realmente el contenido de `x`, simplemente *apunta al* objeto `x`. Así que, si el objeto `x` cambia, apunta a ese nuevo objeto `x`.

Esto puede parecer muy técnico, pero es una característica importante del lenguaje de programación Python. Permite escribir muchos algoritmos distintos de forma más sencilla cuando se pueden modificar listas en su lugar.

Puedes concatenar dos listas sumándolas:

```
x = ['a','b','c']
y = [1,2]
z = x + y
print(z)
```

```
['a', 'b', 'c', 1, 2]
```

También puedes crear una lista repetida mediante multiplicación:

```
x = ['a','b','c']
y = [1]
print(y*3 + x*2)
```

```
[1, 1, 1, 'a', 'b', 'c', 'a', 'b', 'c']
```

Las listas tienen varios *métodos* para modificar su contenido; dos de los más comunes son ordenar e invertir:

```
x = [3,1,2]

x.sort() # sorting the list
print(x)

x.reverse() # reverse ordering
print(x)
```

```
[1, 2, 3]
[3, 2, 1]
```

Note

Cuando ordenas o inviertes una lista, realiza esa operación y modifica la lista *en el lugar*. Así que el código `y = x.sort()` probablemente sea incorrecto, ya que `x.sort()` no devuelve nada. Por lo tanto, `y` no es igual a la lista ordenada, sino que es `None`.

Los métodos son funciones especiales que están vinculadas a objetos específicos. Tienen la forma `object.method(input)`. Siempre estarán separados del objeto base por un punto – esto significa que no puedes usar un punto en un nombre de variable. Por ejemplo, si escribes `q.i = 1` en la terminal, mostrará un error indicando que `q is not defined`. Estos métodos pueden aceptar argumentos adicionales (son como funciones), pero estos ejemplos solo usan los valores predeterminados. Por ejemplo, puedes ordenar en orden descendente:

```
x = [3,1,2]

x.sort(reverse=True) # passing arg
print(x)
```

```
[3, 2, 1]
```

Note

Este es el primer ejemplo que he mostrado de una función que tiene un argumento de *palabra clave*, por lo que en lugar de `function(input)` es `function(keyword=input)`. Las funciones pueden tomar múltiples argumentos, como `function(input1,input2)`. En este escenario el orden de los argumentos importa, y puedes usar argumentos de palabra clave para distinguir entre las entradas. Entraré en más detalles sobre esto en un capítulo posterior sobre cómo definir tus propias funciones.

También puedes agregar o eliminar elementos de listas:

```
x = [3,1,2]

x.append('a') # appending an item to end of list
print(x)

x.remove(1) # removing an item
print(x)
```

```
[3, 1, 2, 'a']
[3, 2, 'a']
```

Para encontrar la ubicación específica de un elemento en una lista, puedes usar el método index:

```
x = ['a','b','c']

# will be 1, the 2nd item in a list
bindex = x.index('b')
print(bindex)
```

```
1
```

Hay otros métodos de las listas que no he mostrado aquí; si ejecutas el comando `dir` en un objeto, mostrará todos sus posibles métodos. Te animo a experimentar por tu cuenta y ver cómo funcionan los demás métodos, como `count` o `pop`.

```
x = ['a','b','c']

# you can look at the methods
# for a object using dir(object)
me = dir(x)

# there are many more! only
# printing a few to save space
print(me[-6:])
```

```
['index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

A veces quieres un objeto similar a una lista, pero no quieres que sea mutable (de modo que no puedas realizar operaciones como cambiar un solo valor, agregar elementos o ordenar la lista). Esto, puede ocurrir si tienes un conjunto de constantes en tu script y sabes que nunca deberían alterarse. Colocarlas en una *tupla* es una forma de asegurarte de que no se modifiquen accidentalmente. Las tuplas se ven casi igual que una lista, pero usan paréntesis en lugar de corchetes:

```
y = (1,2,3)
y[1] = 5 # this will give an error
```

```
TypeError: 'tuple' object does not support item assignment
```

Puede convertir una lista en una tupla mediante el comando `tuple`:

```
y = [1,2,3]
z = tuple(y)
y[1] = 5 # this is ok
```

```
print(y) # can see y list is updated
z[1] = 5 # this is not, again cannot modify tuple
```

```
[1, 5, 3]
```

```
TypeError: 'tuple' object does not support item assignment
```

O, a la inversa, convierta una tupla en una lista mediante el comando `list`:

```
z = (1,2,3)
y = list(z)
y[1] = 5 # this is ok
print(y)
z[1] = 5 # this is not, again cannot modify tuple
```

```
[1, 5, 3]
```

```
TypeError: 'tuple' object does not support item assignment
```

Una última nota sobre las tuplas: puedes asignar varios objetos al mismo tiempo. Así que puedes hacer:

```
x, y = 1, 2
print(x,y)
```

```
1 2
```

Esto se llama *desempaquetado de tuplas*. La razón por la que se llama así es que, cuando *no* se desempaquetan los múltiples valores separados por una coma, se devuelve una tupla.

```
t = 1, 2
print(t)
```

```
(1, 2)
```

Y puedes asignar más de dos valores:

```
a,b,c,d = (1,'a',6,-1.2)
print(a,b,c,d)
```

```
1 a 6 -1.2
```

Esto puede ser conveniente en varios ejemplos de bucles for (como se muestra en el Capítulo 4), y cuando las funciones devuelven múltiples valores (como se muestra en el Capítulo 5). Aparte de esto, sin embargo, en mi experiencia las tuplas no se usan tan comúnmente como las listas. Pero un ejemplo de uso que tienen se ilustra en la siguiente sección, donde se necesita usar tuplas inmutables para diccionarios.

2.5 Diccionarios

El segundo contenedor principal de elementos en python es un diccionario. Así, para acceder a los elementos de una lista, basta con un orden establecido: el primer elemento es `mylist[0]`, el segundo elemento es `mylist[1]`, etc. A veces quieres poder acceder a los elementos mediante nombres más simples; por ejemplo, imagina que tuvieras una lista para contener la información de una persona:

```
# using a list to hold data
x = ['Andy Wheeler','Data Scientist','2019']
```

Para acceder al elemento nombre, necesitas saber que está en el índice 0; el elemento título está en el índice 1, etcétera. Probablemente sea más fácil referirse a estos datos mediante un diccionario.

```
# using a list to hold data
d = {'name': 'Andy Wheeler',
     'title':'Data Scientist',
     'start_year': 2019}

print(d)
```

```
{'name': 'Andy Wheeler', 'title': 'Data Scientist', 'start_year':
2019}
```

Ahora, es más fácil acceder a un elemento individual mediante `dict[key]`, así que si solo quiero el nombre, simplemente lo referencio explícitamente:

```
# Grabbing the specific name element
print(d['name'])
```

```
Andy Wheeler
```

La terminología para los diccionarios es que tienen *claves* que hacen referencia a *valores*. Los valores pueden ser cualquier cosa: valores numéricos, cadenas, listas, otros diccionarios, etc. Sin embargo, las claves deben ser *inmutables*, aunque los diccionarios en sí sean mutables (por lo que no puedes usar una lista como clave, pero sí puedes usar una tupla). Aquí podemos modificar los valores del diccionario original `d` que creé. También puedes añadir un nuevo elemento una vez creado el diccionario.

```
# Modifying the start_year element
d['start_year'] = 2020

# Adding in a new element tenure
d['tenure'] = 3

print(d['name'])
print(d['title'])
print(d['start_year'])
print(d['tenure'])
```

```
Andy Wheeler
Data Scientist
2020
3
```

La mayoría de las veces la gente usa cadenas como claves, ya que el principal beneficio de los diccionarios frente a las listas es tener un nombre para referirse a objetos específicos. Pero también puede ser un número. Así que, si tuvieras identificadores numéricos en otra base de datos que hacen referencia a ubicaciones específicas, podría tener sentido escribir tu diccionario usando esos mismos identificadores numéricos.

```
# You can have numeric values
# as a dictionary key
d = {} # can init a dict as empty
d[101] = {'address': 'Penny Lane', 'tot_crimes': 1}
d[202] = {'address': 'Outer Space', 'tot_crimes': 42}

# These show a dictionary inside of a dictionary
print(d[101])
print(d[202])
```

```
{'address': 'Penny Lane', 'tot_crimes': 1}
{'address': 'Outer Space', 'tot_crimes': 42}
```

Y, al igual que las listas vacías, un diccionario vacío devolverá `False` en una sentencia if booleana:

```
d = {} # can init a dict as empty

if d:
  print('The dictionary d is not empty')
else:
  print('The dictionary d is empty')
```

```
The dictionary d is empty
```

Hay *muchos* tipos de objetos más complicados en python; el primer ejemplo del Índice mostraba cómo trabajar con objetos *datetime*. Pero, bajo el capó, a menudo no son más que contenedores para realizar distintas operaciones con los objetos que enumeré antes: valores numéricos, cadenas, listas y diccionarios.