

O'REILLY® ANAYA
MULTIMEDIA

Actualización 2023

Python

para análisis de datos

Manipulación de datos con pandas, NumPy y Jupyter

Con tecnología



Wes McKinney

Wes McKinney

Python para análisis de datos

Manipulación de datos con pandas, NumPy y Jupyter



Sobre el autor

Wes McKinney es desarrollador de software y empresario en Nashville, Tennessee. Tras obtener su título universitario en matemáticas en el Massachusetts Institute of Technology (MIT) en 2007, empezó a trabajar en finanzas y economía cuantitativa en la compañía AQR Capital Management en Greenwich, Connecticut. Frustrado por las incómodas herramientas de análisis de datos que existían en ese momento, aprendió Python e inició lo que más tarde se convertiría en el proyecto pandas. Es un miembro activo de la comunidad de datos de Python y es defensor del uso de Python en análisis de datos, finanzas y aplicaciones de computación científica.

Posteriormente, Wes fue cofundador y director ejecutivo de DataPad, cuyas instalaciones tecnológicas y personal fueron adquiridos por Cloudera en 2014. Desde entonces ha estado muy implicado en la tecnología Big Data y se ha unido a los comités de administración de los proyectos Apache Arrow y Apache Parquet en la Apache Software Foundation (ASF). En 2018 fundó Usra Labs, una organización sin ánimo de lucro centrada en el desarrollo de Apache Arrow, en asociación con RStudio y Two Sigma Investments. En 2021 ha creado la startup tecnológica Voltron Data, donde trabaja en la actualidad como director de tecnología.

Sobre la imagen de cubierta

El animal de la portada de este libro es una tupaya o musaraña arborícola de cola plumosa (*Ptilocercus lowii*). Este pequeño mamífero es el único de su especie del género *Ptilocercus* y de la familia *Ptiocercidae*; las otras tres musarañas arborícolas que existen son de la familia *Tupaiaidae*. Las tupayas se identifican por sus largas colas y su suave pelo marrón rojizo. Según indica su apodo, la tupaya de cola plumosa tiene una cola que parece una pluma de escribir. Las musarañas de esta clase son omnívoras, se alimentan principalmente de insectos, fruta, semillas y pequeños vertebrados.

Se encuentra principalmente en Indonesia, Malasia y Tailandia, y es conocida por su consumo crónico de alcohol. Se ha descubierto que las musarañas arborícolas de Malasia se pasan varias horas consumiendo el néctar fermentado de forma natural de la palmera de Bertam, lo que equivaldría a unos 10 o 12 vasos de vino con un contenido de alcohol del 3,8 %. A pesar de ello, nunca ninguna tupaya se ha intoxicado, gracias en parte a su impresionante habilidad para descomponer el etanol, que incluye metabolizar el alcohol de una forma no utilizada por los humanos. ¿Algo más impresionante aún que cualquiera de sus colegas mamíferos, incluidos los humanos? La relación entre la masa cerebral y la masa corporal.

A pesar del nombre de estos mamíferos, la tupaya de cola plumosa no es una verdadera musaraña, sino que realmente está más emparentada con los primates. Debido a esta estrecha relación, las musarañas arborícolas se han convertido en una alternativa a los primates en experimentos médicos sobre la miopía, el estrés psicosocial y la hepatitis.

La imagen de la portada procede de la obra *Cassell's Natural History*.

Agradecimientos

Esta obra es producto de muchos años de fructíferas discusiones, colaboraciones y ayuda de muchas personas de todo el mundo. Me gustaría dar las gracias a algunos de ellos.

En memoria de John D. Hunter (1968-2012)

Nuestro estimado amigo y compañero John D. Hunter falleció tras una batalla contra el cáncer de colon el 28 de agosto de 2012, poco después de que yo terminara el manuscrito final para la primera edición de este libro.

Todo lo que se diga acerca del impacto y legado de John en las comunidades científicas y de datos de Python se queda corto. Además de desarrollar matplotlib a principios de los años 2000 (un momento en el que Python apenas era conocido), contribuyó a moldear la cultura de una generación crítica de desarrolladores de código abierto que se habían

convertido en pilares del ecosistema Python que ahora solemos dar por sentado.

Fui lo bastante afortunado como para conectar con John a comienzos de mi carrera con el código abierto en enero de 2010, justo después del lanzamiento de pandas 1.0. Su inspiración y orientación me permitieron mantener firme, incluso en los momentos más oscuros, mi visión de pandas y Python como lenguaje de análisis de primera categoría.

John estaba muy unido a Fernando Pérez y Brian Granger, pioneros de IPython, Jupyter y muchas otras iniciativas de la comunidad Python. Teníamos la esperanza de trabajar los cuatro juntos en un libro, pero yo terminé siendo el que tenía más tiempo libre. Estoy seguro de que se habría sentido orgulloso de lo que hemos logrado, como individuos y como comunidad, en los últimos nueve años.

Agradecimientos por la tercera edición (2022)

Ha pasado más una década desde que empecé a escribir la primera edición de este libro y más de quince años desde que inicié mi viaje como programador de Python. Desde entonces han cambiado muchas cosas. Python ha pasado de ser un lenguaje para análisis de datos relativamente especializado a convertirse en el lenguaje más conocido y utilizado, impulsando así buena parte (¡si no la mayoría!) del trabajo de ciencia de datos, aprendizaje automático e inteligencia artificial.

No he contribuido de forma activa al proyecto pandas de código abierto desde 2013, pero su comunidad internacional de desarrolladores ha seguido progresando y se ha convertido en un modelo de desarrollo de software de código abierto basado en la comunidad. Muchos proyectos Python de «próxima generación» que manejan datos tabulares están creando sus interfaces de usuarios basándose directamente en pandas, de modo que el proyecto ha demostrado tener una influencia perdurable en la futura trayectoria del ecosistema de ciencia de datos de Python.

Espero que este libro siga sirviendo como valioso recurso para estudiantes y para cualquier persona que quiera aprender a trabajar con datos en Python.

Tengo que dar especialmente las gracias a O'Reilly Media por permitirme publicar una versión «de acceso abierto» de este libro en mi sitio web en <https://wesmckinney.com/book>, donde espero que llegue aún a más gente y permita ampliar las oportunidades en el mundo del análisis de datos. J.J. Allaire hizo esto posible siendo mi salvavidas al ayudarme a «transportar» el libro de docbook XML a Quarto (<https://quarto.org>), un fabuloso sistema nuevo de edición científica y técnica para impresión y web.

También quiero dar las gracias a mis revisores técnicos Paul Barry, Jean-Christophe Leyder, Abdullah Karasan y William Jamir, cuyos detallados comentarios han mejorado enormemente la legibilidad, claridad y comprensión del contenido.

Agradecimientos por la segunda edición (2017)

Han pasado casi cinco años desde el día en que terminé el manuscrito para la primera edición de este libro en julio de 2012. En todo este tiempo se han producido muchos cambios. La comunidad Python ha crecido inmensamente, y el ecosistema del software de código abierto que existe a su alrededor se ha fortalecido.

Esta nueva edición del libro no existiría si no fuera por los incansables esfuerzos de los principales desarrolladores de pandas, que han hecho crecer el proyecto y su comunidad de usuarios para convertirlo en uno de los ejes fundamentales del ecosistema de ciencia de datos Python. Entre ellos se incluyen, entre otros, Tom Augspurger, Joris van den Bossche, Chris Bartak, Phillip Cloud, gflyoung, Andy Hayden, Masaaki Horikoshi, Stephan Hoyer, Adam Klein, Wouter Overmeire, Jeff Reback, Chang She, Skipper Seabold, Jeff Tratner e y-p.

En lo referente a la redacción como tal de esta segunda edición, quisiera agradecer al personal de O'Reilly por su paciente ayuda en este proceso. Incluyo a Marie Beaugureau, Ben Lorica y Colleen Toporek. De nuevo disfruté de la ayuda de fabulosos revisores técnicos como Tom Augspurger, Paul Barry, Hugh Brown, Jonathan Coe y Andreas Müller. Muchas gracias.

La primera edición de este libro ha sido traducida a muchos idiomas, incluyendo chino, francés, alemán, japonés, coreano y ruso. Traducir todo este contenido y ponerlo a disposición de una audiencia más amplia es un esfuerzo enorme y con frecuencia no agradecido. Gracias por ayudar a que más personas del mundo aprendan a programar y utilizar herramientas de análisis de datos.

También tengo la suerte de haber recibido apoyo en los últimos años por parte de Cloudera y Two Sigma Investments en mis continuos esfuerzos de desarrollo de código abierto. Teniendo proyectos de software de código abierto con más recursos que nunca en lo que al tamaño de las bases de usuarios se refiere, cada vez está siendo más importante para las empresas ofrecer soporte para el desarrollo de proyectos clave de código abierto. Eso es lo correcto.

Agradecimientos por la primera edición (2012)

Me habría resultado difícil escribir este libro sin el apoyo de un gran grupo de personas.

Del personal de O'Reilly, me siento muy agradecido a mis editores, Meghan Blanchette y Julie Steele, quienes me guiaron en todo el proceso. Mike Loukides trabajó también conmigo en las etapas de la propuesta y ayudó a que el libro se hiciera realidad.

Recibí muchísimas revisiones técnicas de un gran elenco de personajes. En especial, la ayuda de Martin Blais y Hugh Brown resultó increíblemente provechosa para mejorar los ejemplos del libro y su claridad y organización desde la portada hasta la última página. James Long, Drew Conway, Fernando Pérez, Brian Granger, Thomas Kluyver, Adam Klein, Josh Klein, Chang She y Stéfan van der Walt revisaron todos ellos uno o varios capítulos, ofreciendo comentarios acertados desde muchas perspectivas distintas.

Conseguí muchas buenas ideas para los ejemplos y los conjuntos de datos de amigos y compañeros de la comunidad de datos, entre ellos: Mike Dewar, Jeff Hammerbacher, James Johndrow, Kristian Lum, Adam Klein, Hilary Mason, Chang She y Ashley Williams.

Por supuesto, estoy absolutamente en deuda con los líderes de la comunidad científica Python de código abierto que han puesto las bases para mi trabajo de desarrollo y me dieron ánimos mientras escribía este libro: el equipo principal de IPython (Fernando Pérez, Brian Granger, Min Ragan-Kelly, Thomas Kluyver, y otros), John Hunter, Skipper Seabold, Travis Oliphant, Peter Wang, Eric Jones, Robert Kern, Josef Perktold, Francesc Alted, Chris Fonnesbeck y otros tantos que no tengo aquí espacio para mencionar. Otras personas ofrecieron también mucho apoyo, buenas ideas y ánimo en todo el recorrido: Drew Conway, Sean Taylor, Giuseppe Paleologo, Jared Lander, David Epstein, John Krowas, Joshua Bloom, Den Pilsworth, John Myles-White y muchos otros que he olvidado.

También quisiera dar las gracias a una serie de personas a las que conocí en mis años de formación. En primer lugar, mis primeros compañeros de AQR que me dieron ánimos en mi trabajo con pandas a lo largo de los años: Alex Reyfman, Michael Wong, Tim Sargen, Oktay Kurbanov, Matthew Tschantz, Roni Israelov, Michael Katz, Ari Levine, Chris Uga, Prasad Ramanan, Ted Square y Hoon Kim. Por último, gracias a mis consejeros académicos Haynes Miller (MIT) y Mike West (Duke).

Recibí un importante apoyo de Phillip Cloud y Joris van der Bossche en 2014 para actualizar los ejemplos de código del libro y resolver algunas imprecisiones debido a modificaciones sufridas por pandas.

En lo que se refiere a lo personal, Casey me ofreció un apoyo diario inestimable durante el proceso de redacción, tolerando mis altibajos mientras elaboraba el borrador final con una agenda de trabajo ya de por sí sobrecargada. Finalmente, mis padres, Bill y Kim, me enseñaron a seguir siempre mis sueños y nunca conformarme con menos.

Contenido

Sobre el autor

Sobre la imagen de cubierta

Agradecimientos

En memoria de John D. Hunter (1968-2012)

Agradecimientos por la tercera edición (2022)

Agradecimientos por la segunda edición (2017)

Agradecimientos por la primera edición (2012)

Prefacio

Convenciones empleadas en este libro

Uso del código de ejemplo

Capítulo 1. Preliminares

1.1 ¿De qué trata este libro?

¿Qué tipos de datos?

1.2 ¿Por qué Python para análisis de datos?

Python como elemento de unión

Resolver el problema de «los dos lenguajes»

¿Por qué no Python?

1.3 Librerías esenciales de Python

NumPy

pandas

matplotlib

IPython y Jupyter

SciPy

scikit-learn

statsmodels

Otros paquetes

1.4 Instalación y configuración

Miniconda en Windows

GNU/Linux

- Miniconda en macOS
- Instalar los paquetes necesarios
- Entornos de desarrollo integrados y editores de texto
- 1.5 Comunidad y conferencias
- 1.6 Navegar por este libro
 - Códigos de ejemplo
 - Datos para los ejemplos
 - Convenios de importación

Capítulo 2. Fundamentos del lenguaje Python, IPython y Jupyter Notebooks

- 2.1 El intérprete de Python
- 2.2 Fundamentos de IPython
 - Ejecutar el shell de IPython
 - Ejecutar el notebook de Jupyter
 - Autocompletado
 - Introspección
- 2.3 Fundamentos del lenguaje Python
 - Semántica del lenguaje
 - Tipos escalares
 - Control de flujo
- 2.4 Conclusión

Capítulo 3. Estructuras de datos integrados, funciones y archivos

- 3.1 Estructuras de datos y secuencias
 - Tupla
 - Listas
 - Diccionario
 - Conjunto o *set*
 - Funciones de secuencia integradas
 - Comprensiones de lista, conjunto y diccionario
- 3.2 Funciones
 - Espacios de nombres, ámbito y funciones locales
 - Devolver varios valores
 - Las funciones son objetos

- Funciones anónimas (lambda)
- Generadores
- Errores y manejo de excepciones
- 3.3 Archivos y el sistema operativo
 - Bytes y Unicode con archivos
- 3.4 Conclusión

Capítulo 4. Fundamentos de NumPy: arrays y computación vectorizada

- 4.1 El ndarray de NumPy: un objeto array multidimensional
 - Creando ndarrays
 - Tipos de datos para ndarrays
 - Aritmética con arrays NumPy
 - Indexado y corte básicos
 - Indexado booleano
 - Indexado sofisticado
 - Transponer arrays e intercambiar ejes
- 4.2 Generación de números pseudoaleatoria
- 4.3 Funciones universales: funciones rápidas de array elemento a elemento
- 4.4 Programación orientada a arrays con arrays
 - Expresar lógica condicional como operaciones de arrays
 - Métodos matemáticos y estadísticos
 - Métodos para arrays booleanos
 - Ordenación
 - Unique y otra lógica de conjuntos
- 4.5 Entrada y salida de archivos con arrays
- 4.6 Álgebra lineal
- 4.7 Ejemplo: caminos aleatorios
 - Simulando muchos caminos aleatorios al mismo tiempo
- 4.8 Conclusión

Capítulo 5. Empezar a trabajar con pandas

- 5.1 Introducción a las estructuras de datos de pandas
 - Series

- DataFrame
- Objetos índice
- 5.2 Funcionalidad esencial
 - Reindexación
 - Eliminar entradas de un eje
 - Indexación, selección y filtrado
 - Aritmética y alineación de datos
 - Aplicación y asignación de funciones
 - Ordenación y asignación de rangos
 - Índices de ejes con etiquetas duplicadas
- 5.3 Resumir y calcular estadísticas descriptivas
 - Correlación y covarianza
 - Valores únicos, recuentos de valores y pertenencia
- 5.4 Conclusión

Capítulo 6. Carga de datos, almacenamiento y formatos de archivo

- 6.1 Lectura y escritura de datos en formato de texto
 - Leer archivos de texto por partes
 - Escribir datos en formato de texto
 - Trabajar con otros formatos delimitados
 - Datos JSON
 - XML y HTML: raspado web
- 6.2 Formatos de datos binarios
 - Leer archivos de Microsoft Excel
 - Utilizar el formato HDF5
- 6.3 Interactuar con API web
- 6.4 Interactuar con bases de datos
- 6.5 Conclusión

Capítulo 7. Limpieza y preparación de los datos

- 7.1 Gestión de los datos que faltan
 - Filtrado de datos que faltan
 - Rellenado de datos ausentes
- 7.2 Transformación de datos

Eliminación de duplicados

Transformación de datos mediante una función o una asignación

Reemplazar valores

Renombrar índices de eje

Discretización

Detección y filtrado de valores atípicos

Permutación y muestreo aleatorio

Calcular variables dummy o indicadoras

7.3 Tipos de datos de extensión

7.4 Manipulación de cadenas de texto

Métodos de objeto de cadena de texto internos de Python

Expresiones regulares

Funciones de cadena de texto en pandas

7.5 Datos categóricos

Antecedentes y motivación

Tipo de extensión Categorical en pandas

Cálculos con variables categóricas

Métodos categóricos

7.6 Conclusión

Capítulo 8. Disputa de datos: unión, combinación y remodelación

8.1 Indexación jerárquica

Reordenación y clasificación de niveles

Estadísticas de resumen por nivel

Indexación con las columnas de un dataframe

8.2 Combinación y fusión de conjuntos de datos

Uniones de dataframes al estilo de una base de datos

Fusión según el índice

Concatenación a lo largo de un eje

Combinar datos con superposición

8.3 Remodelación y transposición

Remodelación con indexación jerárquica

Transponer del formato «largo» al «ancho»

Transponer del formato «ancho» al «largo»

8.4 Conclusión

Capítulo 9. Gráficos y visualización

9.1 Una breve introducción a la API matplotlib

- Figuras y subgráficos
- Colores, marcadores y estilos de línea
- Marcas, etiquetas y leyendas
- Anotaciones y dibujos en un subgráfico
- Almacenamiento de gráficos en archivo
- Configuración de matplotlib

9.2 Realización de gráficos con pandas y seaborn

- Gráficos de líneas
- Gráficos de barras
- Histogramas y gráficos de densidad
- Gráficos de dispersión o de puntos
- Cuadrícula de facetas y datos categóricos

9.3. Otras herramientas de visualización de Python

9.4 Conclusión

Capítulo 10. Agregación de datos y operaciones con grupos

10.1 Entender las operaciones de grupos

- Iteración a través de grupos
- Selección de una columna o subconjunto de columnas
- Agrupamiento con diccionarios y series
- Agrupamiento con funciones
- Agrupamiento por niveles de índice

10.2 Agregación de datos

- Aplicación de varias funciones a columnas
- Devolución de datos agregados sin índices de fila

10.3 El método apply: un *split-apply-combine* general

- Supresión de las claves de grupos
- Análisis de cuantil y contenedor
- Ejemplo: Rellenar valores faltantes con valores específicos de grupo
- Ejemplo: Muestreo aleatorio y permutación

- Ejemplo: media ponderada de grupo y correlación
- Ejemplo: Regresión lineal por grupos
- 10.4 Transformaciones de grupos y funciones GroupBy «simplificadas»
- 10.5 Tablas dinámicas y tabulación cruzada
 - Tabulaciones cruzadas
- 10.6 Conclusión

Capítulo 11. Series temporales

- 11.1 Tipos de datos de fecha y hora y herramientas asociadas
 - Conversión entre cadena de texto y datetime
- 11.2 Fundamentos de las series temporales
 - Indexación, selección y creación de subconjuntos
 - Series temporales con índices duplicados
- 11.3 Rangos de fechas, frecuencias y desplazamiento
 - Generación de rangos de fechas
 - Frecuencias y desfases de fechas
 - Desplazamiento de los datos (adelantar y retrasar)
- 11.4 Manipulación de zonas horarias
 - Localización y conversión de zonas horarias
 - Operaciones con objetos de marca temporal conscientes de la zona horaria
 - Operaciones entre distintas zonas horarias
- 11.5 Periodos y aritmética de periodos
 - Conversión de frecuencias de periodos
 - Frecuencias de periodos trimestrales
 - Conversión de marcas temporales a periodos (y viceversa)
 - Creación de un objeto PeriodIndex a partir de arrays
- 11.6 Remuestreo y conversión de frecuencias
 - Submuestreo
 - Sobremuestreo e interpolación
 - Remuestreo con periodos
 - Remuestreo de tiempo agrupado
- 11.7 Funciones de ventana móvil
 - Funciones ponderadas exponencialmente

- Funciones binarias de ventana móvil
- Funciones de ventana móvil definidas por el usuario
- 11.8 Conclusión

Capítulo 12. Introducción a las librerías de creación de modelos de Python

- 12.1 Interconexión entre pandas y el código para la creación de modelos
- 12.2 Creación de descripciones de modelos con Patsy
 - Transformaciones de datos en fórmulas Patsy
 - Datos categóricos y Patsy
- 12.3 Introducción a statsmodels
 - Estimación de modelos lineales
 - Estimación de procesos de series temporales
- 12.4 Introducción a scikit-learn
- 12.5 Conclusión

Capítulo 13. Ejemplos de análisis de datos

- 13.1 Datos Bitly de 1.USA.gov
 - Recuento de zonas horarias en Python puro
 - Recuento de zonas horarias con pandas
- 13.2 Conjunto de datos MovieLens 1M
 - Medición del desacuerdo en las valoraciones
- 13.3 Nombres de bebés de Estados Unidos entre 1880 y 2010
 - Análisis de tendencias en los nombres
- 13.4 Base de datos de alimentos del USDA
- 13.5 Base de datos de la Comisión de Elecciones Federales de 2012
 - Estadísticas de donación por ocupación y empleador
 - Incluir donaciones en contenedores
 - Estadísticas de donación por estado
- 13.6 Conclusión

Apéndice A. NumPy avanzado

- A.1 Análisis del objeto ndarray
 - Jerarquía del tipo de datos NumPy
- A.2 Manipulación de arrays avanzada
 - Remodelado de arrays

Orden de C frente a FORTRAN

Concatenación y división de arrays

Repetición de elementos: tile y repeat

Equivalentes del indexado sofisticado: take y put

A.3 Difusión

Difusión a lo largo de otros ejes

Configuración de valores de array por difusión

A.4 Uso avanzado de ufuncs

Métodos de instancia ufunc

Escribir nuevas ufuncs en Python

A.5 Arrays estructurados y de registros

Tipos de datos anidados y campos multidimensionales

¿Por qué emplear arrays estructurados?

A.6 Más sobre la ordenación

Ordenaciones indirectas: argsort y lexsort

Algoritmos de ordenación alternativos

Ordenación parcial de arrays

Localización de elementos en un array ordenado con
numpy.searchsorted

A.7 Escritura de funciones rápidas NumPy con Numba

Creación de objetos personalizados numpy.ufunc con Numba

A.8 Entrada y salida de arrays avanzadas

Archivos mapeados en memoria

HDF5 y otras opciones de almacenamiento de arrays

A.9 Consejos de rendimiento

La importancia de la memoria contigua

Apéndice B. Más sobre el sistema IPython

B.1 Atajos de teclado del terminal

B.2 Los comandos mágicos

El comando %run

Ejecutar código desde el portapapeles

B.3 Cómo utilizar el historial de comandos

Búsqueda y reutilización del historial de comandos

Variables de entrada y salida

B.4 Interacción con el sistema operativo

Comandos de shell y alias

Sistema de marcado a directorios

B.5 Herramientas de desarrollo de software

Depurador interactivo

Medir el tiempo de ejecución del código: `%time` y `%timeit`

Perfilado básico: `%prun` y `%run -p`

Perfilar una función línea a línea

B.6 Consejos para un desarrollo de código productivo con IPython

Recargar dependencias de módulo

Consejos de diseño de código

B.7 Funciones avanzadas de IPython

Perfiles y configuración

B.8 Conclusión

Créditos

Prefacio

La primera edición de este libro se publicó en 2012, en una época en la que las librerías de análisis de datos de fuente abierta de Python, especialmente pandas, eran nuevas y se estaban desarrollando a gran velocidad. Cuando llegó el momento de escribir la segunda edición en 2016 y 2017, necesité actualizar el libro no solo para Python 3.6 (la primera edición empleaba Python 2.7), sino también para los abundantes cambios producidos en pandas en los cinco años anteriores. Ahora, en 2022, hay menos cambios en el lenguaje Python (estamos ya en Python 3.10, con la versión 3.11 a punto de llegar a finales de 2022), pero pandas ha seguido evolucionando.

En esta tercera edición, mi objetivo es actualizar el contenido con las versiones actuales de Python, NumPy, pandas y otros proyectos, manteniéndome al mismo tiempo relativamente conservador en lo relativo a los proyectos Python más recientes surgidos en los últimos años. Como este libro se ha convertido en un recurso de gran importancia para muchos cursos universitarios y profesionales del sector, trataré de evitar temas que puedan quedar obsoletos en un año o dos. De esa forma, las copias en papel no resultarán demasiado difíciles de seguir en 2023, 2024 o más allá.

Una nueva característica de la tercera edición es la versión en línea de acceso abierto alojada en mi sitio web en <https://wesmckinney.com/book>, que sirve como recurso y resulta cómodo para poseedores de las ediciones impresa y digital. Trato de mantener ahí el contenido razonablemente actualizado, de modo que si dispone de una copia en papel y se encuentra con algo que no funciona correctamente, recomiendo revisar en mi web los últimos cambios en el contenido.

Convenciones empleadas en este libro

En este libro se utilizan las siguientes convenciones tipográficas:

- *Cursiva*: Es un tipo que se usa para diferenciar términos anglosajones o de uso poco común. También se usa para destacar algún concepto.
- **Negrita**: Le ayudará a localizar rápidamente elementos como las combinaciones de teclas.
- Fuente especial: Nombres de botones y opciones de programas. Por ejemplo, Aceptar para hacer referencia a un botón con ese título.
- Monoespacial: Utilizado para el código y dentro de los párrafos para hacer referencia a elementos como nombres de variables o funciones, bases de datos, tipos de datos, variables de entorno, declaraciones y palabras clave.
- También encontrará a lo largo del libro recuadros con elementos destacados sobre el texto normal, para comunicarle de manera breve y rápida algún concepto relacionado con lo que está leyendo.



Este elemento representa un truco o una sugerencia.



Este elemento representa una nota.



Este elemento representa una advertencia o precaución.

Uso del código de ejemplo

Se puede descargar material adicional (ejemplos de código, ejercicios, etc.) de la página web de Anaya Multimedia

(<http://www.anayamultimedia.es>). Vaya al botón Selecciona Complemento de la ficha del libro, donde podrá descargar el contenido para utilizarlo directamente. También puede descargar el material de la página web original del libro (<https://github.com/wesm/pydata-book>), que está duplicado en Gitee (para quienes no puedan acceder a GitHub) en <https://gitee.com/wesmckinn/pydata-book>.

Este libro ha sido creado para ayudarle en su trabajo. En general, puede utilizar el código de ejemplo ofrecido en este libro en sus programas y en su documentación. No es necesario contactar con nosotros para solicitar permiso, a menos que esté reproduciendo una gran cantidad del código. Por ejemplo, escribir un programa que utilice varios fragmentos de código tomados de este libro no requiere permiso. Sin embargo, vender o distribuir ejemplos de los libros de O'Reilly sí lo requiere. Responder una pregunta citando este libro y empleando textualmente código de ejemplo incluido en él no requiere permiso. Pero incorporar una importante cantidad de código de ejemplo de este libro en la documentación de su producto sí lo requeriría.

1.1 ¿De qué trata este libro?

Este libro se ocupa de los aspectos prácticos de manipular, procesar, limpiar y desmenuzar datos en Python. El objetivo es ofrecer una guía de los componentes del lenguaje de programación Python y su ecosistema de librerías y herramientas orientadas a datos, que permita al lector equiparse para convertirse en un analista de datos efectivo. Aunque «análisis de datos» forma parte del título del libro, el objetivo específico del mismo es la programación de Python y sus librerías y herramientas, a diferencia de la metodología del análisis de datos. Esta es la programación de Python que necesita para análisis de datos.

En algún momento posterior a la publicación de este libro en 2012, se empezó a utilizar el término «ciencia de datos» como una descripción general para todo, desde sencillas estadísticas descriptivas hasta análisis estadísticos más avanzados y aprendizaje automático. El ecosistema de código abierto de Python para hacer análisis de datos (o ciencia de datos) también se ha expandido notablemente desde entonces. Ahora hay muchos otros libros que se centran concretamente en estas metodologías más avanzadas. Confío en que este libro sirva como preparación adecuada para permitir a sus lectores avanzar a un recurso de dominio más específico.



Quizá haya gente que describa buena parte del contenido del libro como «manipulación de datos» a diferencia de «análisis de datos». También emplearemos los términos «disputa» (*wrangling*) o «procesado» (*munging*) para referirnos a la manipulación de datos.

¿Qué tipos de datos?

Cuando decimos «datos», ¿a qué nos referimos exactamente? El principal enfoque se centra en datos estructurados, un término deliberadamente genérico que abarca muchas formas comunes de datos, como por ejemplo:

- Datos tabulares o en forma de hoja de cálculo, en los que cada columna puede ser de un tipo distinto (cadena de texto, numérico, fecha u otro). Incluye la mayoría de los tipos de datos almacenados normalmente en bases de datos relacionales o en archivos de texto delimitados por tabuladores o comas.
- Arrays multidimensionales (matrices).

- Tablas múltiples de datos interrelacionados por columnas clave (lo que serían claves primarias o externas para un usuario de SQL).
- Series temporales espaciadas uniformemente o de manera desigual.

Sin duda, esta no es una lista completa. Aunque no siempre pueda ser obvio, a un gran porcentaje de conjuntos de datos se le puede dar una forma estructurada, más adecuada para análisis y modelado de datos. Si no, puede ser posible extraer características de un conjunto de datos para darles una forma estructurada. Como ejemplo, se podría procesar una colección de artículos de prensa hasta convertirla en una tabla de frecuencia de palabras, que se puede emplear después para realizar análisis de opiniones.

A la mayoría de los usuarios de programas de hoja de cálculo como Microsoft Excel, quizá la herramienta de análisis de datos más utilizada en todo el mundo, no les resultarán raros estos tipos de datos.

1.2 ¿Por qué Python para análisis de datos?

Para muchos, el lenguaje de programación Python tiene un gran atractivo. Desde su primera aparición en 1991, Python se ha convertido en uno de los lenguajes de programación de intérprete más conocidos, junto con Perl, Ruby y otros. Python y Ruby se han hecho especialmente populares desde 2005 más o menos por crear sitios web utilizando sus diferentes *frameworks* web, como Rails (Ruby) y Django (Python). A estos lenguajes se les llama lenguajes de *scripting* o secuencia de comandos, pues se pueden emplear para escribir rápidamente programas —o secuencias de comandos— de poca entidad para automatizar otras tareas. No me gusta el término «lenguaje de secuencia de comandos», porque lleva consigo la connotación de que no se puede utilizar para crear software serio. De entre los lenguajes interpretados, por distintas razones históricas y culturales, Python ha desarrollado una comunidad de análisis de datos y computación científica muy grande y activa. En los últimos 20 años, Python ha pasado de ser un lenguaje de ciencia computacional de vanguardia, es decir, «bajo tu cuenta y riesgo», a uno de los lenguajes más importantes para la ciencia de datos, el aprendizaje automático y el desarrollo general de software, tanto académicamente hablando como dentro del sector.

Para análisis de datos, computación interactiva y visualización de datos, es inevitable que Python dé lugar a comparaciones con otros lenguajes de programación y herramientas de fuente abierta y comerciales de uso generalizado, como R, MATLAB, SAS, Stata, etc. En los últimos años, las librerías de código abierto mejoradas de Python (como pandas y scikit-learn) lo han convertido en la opción habitual para tareas de análisis de datos. Combinadas con la solidez global de Python para ingeniería de software genérica, es una excelente alternativa como lenguaje principal para crear aplicaciones de datos.

Python como elemento de unión

Parte del éxito de Python en la ciencia computacional se debe a la facilidad de integración de código de C, C++ y FORTRAN. La mayoría de los entornos de computación modernos comparten un conjunto parecido de librerías de FORTRAN y C heredadas para realizar algoritmos de álgebra lineal, optimización, integración, transformadas de Fourier rápidas y otros similares. La misma historia se aplica a muchas empresas y laboratorios que han utilizado Python para aglutinar décadas de software heredado.

Muchos programas están formados por pequeños fragmentos de código en los que se invierte la mayor parte del tiempo, porque contienen grandes cantidades de «código de pegamento» que con frecuencia no funcionan. En muchos casos, el tiempo de ejecución del código de pegamento es insignificante; la mayor parte del esfuerzo se invierte de manera fructífera en optimizar los atascos computacionales, en ocasiones traduciendo el código a un lenguaje de menor nivel como C.

Resolver el problema de «los dos lenguajes»

En muchas compañías, es habitual investigar, crear prototipos y probar nuevas ideas utilizando un lenguaje de programación más especializado como SAS o R, y después trasladar esas ideas para que formen parte de un sistema de producción mayor escrito en, por ejemplo, Java, C# o C++. Lo que la gente está descubriendo poco a poco es que Python es un lenguaje adecuado no solo para realizar investigaciones y prototipos, sino también para crear los sistemas de producción. ¿Por qué mantener dos entornos de desarrollo cuando con uno basta? Creo que cada vez más empresas van a seguir este camino, porque tener investigadores e ingenieros de software que utilicen el mismo conjunto de herramientas de programación proporciona muchas veces importantes beneficios para la organización.

Durante la última década han surgido nuevos enfoques destinados a resolver el problema de «los dos lenguajes», por ejemplo, el lenguaje de programación Julia. Sacar lo mejor de Python requerirá en muchos casos programar en un lenguaje de bajo nivel como C o C++ y crear vinculaciones de Python con ese código. Dicho esto, la tecnología de compilación JIT («just in time»: justo a tiempo), ofrecida por librerías como Numba, ha supuesto una forma de lograr un excelente rendimiento en muchos algoritmos computacionales sin tener que abandonar el entorno de programación de Python.

¿Por qué no Python?

Aunque Python es un excelente entorno para crear muchos tipos de aplicaciones analíticas y sistemas de propósito general, hay muchos aspectos en los que Python puede no ser tan útil.

Como Python es un lenguaje de programación interpretado, en general la mayor parte del código Python se ejecutará notablemente más despacio que otro código que haya sido escrito en un lenguaje compilado como Java o C++. Como el tiempo de programación suele ser más valioso que el tiempo de CPU, para muchos este cambio es ideal. No obstante, en una aplicación con latencia muy baja o requisitos de uso de recursos muy exigentes (por ejemplo, un sistema de comercio de alta frecuencia), el tiempo empleado programando en un lenguaje de bajo nivel (pero también de baja productividad), como C++, para lograr el máximo rendimiento posible podría ser tiempo bien empleado.

Python puede ser un lenguaje complicado para crear aplicaciones multitarea de alta concurrencia, especialmente las que tienen muchas tareas ligadas a la CPU. La razón de esto es que tiene lo que se conoce como GIL (*Global Interpreter Lock*), o bloqueo de intérprete global, un mecanismo que evita que el intérprete ejecute más de una instrucción de Python al mismo tiempo. Las razones técnicas de la existencia de GIL quedan fuera del alcance de este libro. Aunque es cierto que en muchas aplicaciones de procesamiento de *big data* puede ser necesario un grupo de ordenadores para procesar un conjunto de datos en un espacio de tiempo razonable, siguen existiendo situaciones en las que es preferible un sistema multitarea de un solo proceso.

Esto no significa que Python no pueda ejecutar código paralelo multitarea. Las extensiones en C de Python que emplean multitarea nativa (en C o C++) pueden ejecutar código en paralelo sin verse afectadas por el bloqueo GIL, siempre que no necesiten interactuar regularmente con objetos Python.

1.3 Librerías esenciales de Python

Para todos aquellos que no estén familiarizados con el ecosistema de datos de Python y las librerías empleadas a lo largo de este libro, aquí va un breve resumen de algunas de ellas.

NumPy

NumPy (<https://numpy.org>), abreviatura de *Numerical Python* (Python numérico), ha sido durante mucho tiempo la piedra angular de la computación numérica en Python. Ofrece las estructuras de datos, los algoritmos y el «pegamento» necesario para la mayoría de las aplicaciones científicas que tienen que ver con datos numéricos en Python. NumPy contiene, entre otras cosas:

- Un objeto array *ndarray* multidimensional, rápido y eficaz.
- Funciones para realizar cálculos por elementos con arrays u operaciones matemáticas entre arrays.
- Herramientas para leer y escribir en disco conjuntos de datos basados en arrays.

- Operaciones de álgebra lineal, transformadas de Fourier y generación de números aleatorios.
- Una API de C muy desarrollada que permite a las extensiones de Python y al código C o C++ nativo acceder a las estructuras de datos y a las utilidades computacionales de NumPy.

Más allá de las rápidas habilidades de proceso de arrays que NumPy le incorpora a Python, otro de sus usos principales en análisis de datos es como contenedor para pasar datos entre algoritmos y librerías. Para datos numéricos, los arrays de NumPy son más eficaces para almacenar y manipular datos que las otras estructuras de datos integradas en Python. Además, las librerías escritas en un lenguaje de bajo nivel, como C o FORTRAN, pueden trabajar con los datos almacenados en un array de NumPy sin copiar datos en otra representación de memoria distinta. De esta forma, muchas herramientas de cálculo numérico para Python, o bien admiten los arrays de NumPy como estructura de datos principal, o bien se centran en la interoperabilidad con NumPy.

pandas

pandas (<https://pandas.pydata.org>) ofrece estructuras de datos y funciones de alto nivel diseñadas para flexibilizar el trabajo con datos estructurados o tabulares. Desde su nacimiento en 2010, ha reforzado a Python como un potente y productivo entorno de análisis de datos. Los objetos principales de pandas que se utilizarán en este libro son DataFrame, una estructura de datos tabular y orientada a columnas con etiquetas de fila y columna, y Series, un objeto array con etiquetas y unidimensional.

La librería pandas fusiona las ideas de NumPy sobre cálculo de arrays con el tipo de habilidades de manipulación de datos que se pueden encontrar en hojas de cálculo y bases de datos relacionales (como SQL). Ofrece una cómoda funcionalidad de indexado para poder redimensionar, segmentar, realizar agregaciones y seleccionar subconjuntos de datos. Como la manipulación, preparación y limpieza de los datos es una habilidad tan importante en análisis de datos, pandas es uno de los focos de atención principales de este libro.

Para poner al lector en antecedentes, empecé a crear pandas a principios de 2008, durante el tiempo que estuve en AQR Capital Management, una empresa de administración de inversiones cuantitativas. En aquel momento, tenía una serie de requisitos muy claros que no estaban siendo bien resueltos por ninguna herramienta de las que tenía a mi disposición:

- Estructuras de datos con ejes etiquetados que soporten alineación de datos automática o explícita (lo que evita errores habituales, resultado de datos mal alineados, e impide trabajar con datos indexados procedentes de distintas fuentes).
- Funcionalidad integrada de series temporales.

- Las mismas estructuras de datos manejan datos de series temporales e intemporales.
- Operaciones aritméticas y reducciones que conserven los metadatos.
- Manejo flexible de datos faltantes.
- Operación de unión y otras operaciones relacionales de bases de datos conocidas (basadas en SQL, por ejemplo).

Mi idea era poder hacer todas estas cosas de una sola vez, preferiblemente en un lenguaje adecuado para el desarrollo de software genérico. Python era un buen candidato para ello, pero en ese momento no había un conjunto integrado de estructuras de datos y herramientas que ofrecieran esta funcionalidad. Como resultado de haber sido creado inicialmente para resolver problemas analíticos financieros y empresariales, pandas cuenta con una funcionalidad de series temporales especialmente profunda y con herramientas idóneas para trabajar con datos indexados en el tiempo y generados por procesos empresariales.

Me pasé buena parte de 2011 y 2012 ampliando las habilidades de pandas con la ayuda de dos de mis primeros compañeros de trabajo de AQR, Adam Klein y Chang She. En 2013, dejé de estar tan implicado en el desarrollo diario de proyectos, y desde entonces pandas se ha convertido en un proyecto propiedad por completo de su comunidad y mantenido por ella, con más de 2000 colaboradores únicos en todo el mundo.

A los usuarios del lenguaje R de cálculos estadísticos, el nombre `DataFrame` les resultará familiar, ya que el objeto se denominó así por el objeto `data.frame` de R. A diferencia de Python, los marcos de datos o *data frames* están integrados en el lenguaje de programación R y en su librería estándar. Como resultado de ello, muchas funciones de pandas suelen ser parte de la implementación esencial de R, o bien son proporcionadas por paquetes adicionales.

El propio nombre pandas deriva de *panel data*, un término de econometría que define conjuntos de datos estructurados y multidimensionales, y también un juego de palabras con la expresión inglesa «*Python data analysis*» (análisis de datos de Python).

matplotlib

matplotlib (<https://matplotlib.org>) es la librería de Python más conocida para producir gráficos y otras visualizaciones de datos bidimensionales. Fue creada originalmente por John D. Hunter, y en la actualidad un nutrido equipo de desarrolladores se encarga de mantenerla. Fue diseñada para crear gráficos adecuados para su publicación. Aunque hay otras librerías de visualización disponibles para programadores Python, matplotlib sigue siendo muy utilizada y se integra razonablemente bien con el resto del ecosistema. Creo que es una opción segura como herramienta de visualización predeterminada.

IPython y Jupyter

El proyecto IPython (<https://ipython.org>) se inició en 2001 como proyecto secundario de Fernando Pérez para crear un mejor intérprete de Python interactivo. En los siguientes 20 años se ha convertido en una de las herramientas más importantes de la moderna pila de datos de Python. Aunque no ofrece por sí mismo herramientas computacionales o para análisis de datos, IPython se ha diseñado tanto para desarrollo de software como para computación interactiva. Aboga por un flujo de trabajo ejecución-exploración, en lugar del típico flujo editar-compilar-ejecutar de muchos otros lenguajes de programación. También proporciona acceso integrado al shell y al sistema de archivos del sistema operativo de cada usuario, lo que reduce la necesidad de cambiar entre una ventana de terminal y una sesión de Python en muchos casos. Como buena parte de la codificación para análisis de datos implica exploración, prueba y error, además de repetición, IPython puede lograr que todo este trabajo se haga mucho más rápido.

En 2014, Fernando y el equipo de IPython anunciaron el proyecto Jupyter (<https://jupyter.org>), una iniciativa más amplia para diseñar herramientas de computación interactiva para cualquier tipo de lenguaje. El notebook (cuaderno) de IPython basado en la web se convirtió en Jupyter Notebook, que ahora dispone de soporte de más de 40 lenguajes de programación. El sistema IPython puede emplearse ahora como *kernel* (un modo de lenguaje de programación) para utilizar Python con Jupyter. El propio IPython se ha convertido en un componente del proyecto de fuente abierta Jupyter mucho más extenso, que ofrece un entorno productivo para computación interactiva y exploratoria. Su «modo» más antiguo y sencillo es un shell de Python diseñado para acelerar la escritura, prueba y depuración del código Python. También se puede usar el sistema IPython a través de Jupyter Notebook.

El sistema Jupyter Notebook también permite crear contenidos en Markdown y HTML y proporciona un medio para crear documentos enriquecidos con código y texto.

Personalmente, yo utilizo IPython y Jupyter habitualmente en mi trabajo con Python, ya sea ejecutando, depurando o probando código.

En el material contenido en GitHub que acompaña al libro (<https://github.com/wesm/pydata-book>) se podrán encontrar notebooks de Jupyter que contienen todos los ejemplos de código de cada capítulo. Si no es posible acceder a GitHub, se puede probar el duplicado en Gitee (<https://gitee.com/wesmckinn/pydata-book>).

SciPy

SciPy (<https://scipy.org>) es una colección de paquetes que resuelve una serie de problemas de base en la ciencia computacional. Estas son algunas de las herramientas que contienen sus distintos módulos:

- `scipy.integrate`: Rutinas de integración numéricas y distintos resolutores de ecuaciones.
- `scipy.linalg`: Rutinas de álgebra lineal y descomposiciones de matrices que van más allá de los proporcionados por `numpy.linalg`.
- `scipy.optimize`: Optimizadores (minimizadores) de funciones y algoritmos de búsqueda de raíces.
- `scipy.signal`: Herramientas de procesamiento de señal.
- `scipy.sparse`: Matrices dispersas y resolutores de sistemas lineales dispersos.
- `scipy.special`: Contenedor de SPECFUN, una librería de FORTRAN que implementa muchas funciones matemáticas comunes, como la función `gamma`.
- `scipy.stats`: Distribuciones de probabilidad estándares continuas y discretas (funciones de densidad, muestreadores, funciones de distribución continua), diversas pruebas estadísticas y más estadísticas descriptivas.

NumPy y SciPy, juntos, forman una base computacional razonablemente completa y desarrollada para muchas aplicaciones tradicionales de ciencia computacional.

scikit-learn

Desde los inicios del proyecto en 2007, `scikit-learn` (<https://scikit-learn.org>) se ha convertido en el principal juego de herramientas de aprendizaje automático de uso general para programadores de Python. En el momento de escribir esto, más de 2000 personas han contribuido con código al proyecto. Incluye submódulos para modelos como:

- Clasificación: SVM, vecinos más cercanos, bosque aleatorio, regresión logística, etc.
- Regresión: Lasso, regresión *ridge*, etc.
- Agrupamiento (*clustering*): *k-means*, agrupamiento espectral, etc.
- Reducción de dimensionalidad: PCA, selección de características, factorización de matrices, etc.
- Selección de modelo: búsqueda en rejilla, validación cruzada, métricas.
- Preprocesamiento: extracción de características, normalización.

Junto con `pandas`, `statsmodels` e `IPython`, `scikit-learn` ha sido fundamental para convertir a Python en un productivo lenguaje de programación de ciencia de datos. Aunque no pueda incluir en este libro una guía completa de `scikit-learn`, sí puedo ofrecer una breve introducción de algunos de sus modelos y explicar cómo utilizarlos con las otras herramientas presentadas aquí.

statsmodels

statsmodels (<https://statsmodels.org>) es un paquete de análisis estadístico que germinó gracias al trabajo de Jonathan Taylor, profesor de estadística de la Universidad de Stanford, quien implementó una serie de modelos de análisis de regresión conocidos en el lenguaje de programación R. Skipper Seabold y Josef Perktold crearon formalmente el nuevo proyecto statsmodels en 2010, y desde entonces han hecho crecer el proyecto hasta convertirlo en una masa ingente de usuarios y colaboradores comprometidos. Nathaniel Smith desarrolló el proyecto Patsy, que ofrece un marco de especificaciones de fórmulas o modelos para statsmodels inspirado en el sistema de fórmulas de R.

Comparado con scikit-learn, statsmodels contiene algoritmos para estadística clásica (principalmente frecuentista) y econometría, que incluyen submódulos como:

- Modelos de regresión: regresión lineal, modelos lineales generalizados, modelos lineales robustos, modelos lineales mixtos, etc.
- Análisis de varianza (ANOVA).
- Análisis de series temporales: AR, ARMA, ARIMA, VAR y otros modelos.
- Métodos no paramétricos: estimación de densidad de kernel, regresión de kernel.
- Visualización de resultados de modelos estadísticos.

statsmodels se centra más en la inferencia estadística, ofreciendo estimación de incertidumbres y valores p para parámetros. scikit-learn, por el contrario, está más enfocado en la predicción.

Como con scikit-learn, ofreceré una breve introducción a statsmodels y explicaré cómo utilizarlo con NumPy y pandas.

Otros paquetes

Ahora, en 2022, hay muchas otras librerías de Python de las que se podría hablar en un libro sobre ciencia de datos. Entre ellas se incluyen varios proyectos de reciente creación, como TensorFlow o PyTorch, que se han hecho populares para trabajar con aprendizaje automático o inteligencia artificial. Ahora que ya hay otros libros en el mercado que tratan específicamente esos proyectos, yo recomendaría utilizar este libro para crear una buena base en manipulación de datos genérica en Python. Tras su lectura, es muy probable que ya se esté bien preparado para pasar a un recurso más avanzado que pueda presuponer un cierto nivel de experiencia.

1.4 Instalación y configuración

Como todo el mundo utiliza Python para distintas aplicaciones, no hay una solución única para configurar Python y obtener los paquetes adicionales necesarios. Es probable que muchos lectores no tengan un completo entorno de desarrollo Python adecuado para poder seguir este libro, de modo que voy a dar instrucciones detalladas para configurar cada sistema operativo. Utilizaré Miniconda, una instalación mínima del administrador de paquetes conda, además de conda-forge (<https://conda-forge.org>), una distribución de software mantenida por la comunidad y basada en conda. Este libro trabaja con Python 3.10, pero si alguno de mis lectores lo está leyendo en el futuro, puede instalar perfectamente una versión más reciente de Python.

Si por alguna razón estas instrucciones se quedan obsoletas para cuando esté leyendo esto, puede consultar el libro en mi sitio web (<https://wesmckinney.com/book>), que me esforzaré por mantener actualizado con las instrucciones de instalación más recientes.

Miniconda en Windows

Para empezar en Windows, descargue el instalador de Miniconda para la última versión de Python disponible (ahora mismo 3.9) de la página <https://conda.io>. Recomiendo seguir las instrucciones de instalación para Windows disponibles en el sitio web de conda, que quizá hayan cambiado entre el momento en que se publicó este libro y el momento en el que esté leyendo esto. La mayoría de la gente querrá la versión de 64 bits, pero si no funciona en su máquina Windows, puede instalar sin problemas la versión de 32 bits.

Cuando le pregunten si desea realizar la instalación solo para usted o para todos los usuarios de su sistema, elija la opción más adecuada para usted. La instalación individual bastará para seguir el libro. También le preguntarán si desea añadir Miniconda a la variable de entorno PATH del sistema. Si dice que sí (yo normalmente lo hago), entonces esta instalación de Miniconda podría anular otras versiones de Python que pudiera tener instaladas. Si contesta que no, entonces tendrá que utilizar el atajo del menú de inicio de Windows que se haya instalado para poder utilizar este Miniconda. Dicha entrada podría llamarse algo así como «Anaconda3 (64-bit)».

Supondré que no ha añadido Miniconda al PATH de su sistema. Para verificar que las cosas estén correctamente configuradas, abra la entrada «Anaconda Prompt (Miniconda3)» dentro de «Anaconda3 (64-bit)» en el menú Inicio. A continuación, intente lanzar el intérprete Python escribiendo **python**. Debería aparecer un mensaje como este:

```
(base) C:\Users\Wes>python
Python 3.9 [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.

>>>
```

Para salir del shell de Python, escriba el comando **exit()** y pulse **Intro**.

GNU/Linux

Los detalles de Linux variarán dependiendo del tipo de distribución Linux que se tenga; aquí daré información para distribuciones como Debian, Ubuntu, CentOS y Fedora. La configuración es similar a la de macOS, con la excepción de cómo esté instalado Miniconda. La mayoría de los lectores descargarán el archivo instalador de 64 bits predeterminado, que es para arquitectura x86 (pero es posible que en el futuro más usuarios tengan máquinas Linux basadas en aarch64). El instalador es un *shell-script* que se debe ejecutar en el terminal. Entonces dispondrá de un archivo con un nombre parecido a `Miniconda3-latest-Linux-x86_64.sh`. Para instalarlo, ejecute este fragmento de código con `bash`:

```
$ bash Miniconda3-latest-Linux-x86_64.sh
```



Ciertas distribuciones de Linux incluirán en sus administradores todos los paquetes de Python necesarios (aunque versiones obsoletas, en algunos casos), y se pueden instalar usando una herramienta como `apt`. La configuración descrita aquí utiliza Miniconda, ya que se puede reproducir mucho más fácilmente en las distintas distribuciones y resulta más sencillo actualizar paquetes a sus versiones más recientes.

Le presentarán una selección de opciones para colocar los archivos de Miniconda. Yo recomiendo instalar los archivos en la ubicación predeterminada de su directorio de inicio, por ejemplo `/inicio/$USUARIO/miniconda` (con su nombre de usuario, naturalmente).

El instalador le preguntará si desea modificar los *scripts* del shell para activar automáticamente Miniconda. Yo le recomiendo que lo haga (diga “sí”) por una simple cuestión de comodidad.

Tras completar la instalación, inicie un nuevo proceso de terminal y verifique que está seleccionando la nueva instalación de Miniconda:

```
(base) $ python
Python 3.9 | (main) [GCC 10.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>>
```

Para salir del shell de Python, teclee **exit()** y pulse **Intro** o **Control-D**.

Miniconda en macOS

Descargue el instalador de Miniconda para macOS, cuyo nombre debería ser algo así como `Miniconda3-latest-MacOSX-arm64.sh` para ordenadores macOS con Apple

Silicon lanzados del 2020 en adelante, o bien `Miniconda3-latest-MacOSX-x86_64.sh` para Macs con Intel lanzados antes de 2020. Abra la aplicación Terminal de macOS e instale ejecutando el instalador (lo más probable en su directorio de descargas) con `bash`.

```
$ bash $HOME/Downloads/Miniconda3-latest-MacOSX-arm64.sh
```

Cuando se ejecute el instalador, configurará automáticamente por defecto Miniconda en su entorno y perfil shell predeterminados, probablemente en `/Usuarios/$USUARIO/.zshrc`. Le recomiendo dejar que lo haga así; si no desea permitir que el instalador modifique su entorno de shell predeterminado, tendrá que consultar la documentación de Miniconda para saber cómo continuar.

Para verificar que todo funcione correctamente, intente lanzar Python en el shell del sistema (abra la aplicación Terminal para obtener una línea de comandos):

```
$ python
Python 3.9 (main) [Clang 12.0.1 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Para salir del shell, pulse **Control-D** o teclee `exit()` y pulse **Intro**.

Instalar los paquetes necesarios

Ahora que ya está Miniconda configurado en su sistema, es hora de instalar los principales paquetes que utilizaremos en este libro. El primer paso es configurar conda-forge como canal de paquetes predeterminado ejecutando los siguientes comandos en un shell:

```
(base) $ conda config --add channels conda-forge
(base) $ conda config --set channel_priority strict
```

Ahora crearemos un nuevo “entorno” conda mediante el comando `conda create` que utiliza Python 3.10:

```
(base) $ conda create -y -n pydata-book python=3.10
```

Una vez terminada la instalación, active el entorno con `conda activate`:

```
(base) $ conda activate pydata-book
(pydata-book) $
```



Es necesario utilizar `conda activate` para activar el entorno cada vez que se abra un nuevo terminal. Puede ver información sobre el entorno activo de conda en cualquier momento desde el terminal utilizando el comando `conda info`.

A continuación instalaremos los paquetes esenciales empleados a lo largo del libro (junto con sus dependencias) con `conda install`:

```
(pydata-book) $ conda install -y pandas jupyter matplotlib
```

Utilizaremos también otros paquetes, pero pueden instalarse más tarde, cuando sean necesarios. Hay dos formas de instalar paquetes, con `conda install` y con `pip install`. Siempre es preferible `conda install` al trabajar con Miniconda, pero algunos paquetes no están disponibles en conda, de modo que si `conda install $nombre_paquete` falla, pruebe con `pip install $nombre_paquete`.



Si quiere instalar todos los paquetes utilizados en el resto del libro, puede hacerlo ya ejecutando:

```
conda install lxml beautifulsoup4 html5lib openpyxl \
requests sqlalchemy seaborn scipy statsmodels \
patsy scikit-learn pyarrow pytables numba
```

En Windows, para indicar la continuación de línea ponga un carácter `^` en lugar de la barra invertida `\` empleada en Linux y macOS.

Puede actualizar paquetes utilizando el comando `conda update`:

```
conda update nombre_paquete
```

`pip` soporta también actualizaciones usando la bandera `-upgrade`:

```
pip install -upgrade nombre_paquete
```

Tendrá variadas oportunidades de probar estos comandos a lo largo del libro.



Aunque se pueden utilizar tanto `conda` como `pip` para instalar paquetes, conviene evitar actualizar paquetes instalados originalmente con `conda` utilizando `pip` (y viceversa), ya que hacer esto puede dar lugar a problemas en el entorno. Recomiendo quedarse con `conda` si es posible, volviendo a `pip` solo para paquetes que no estén disponibles con `conda install`.

Entornos de desarrollo integrados y editores de texto

Cuando me preguntan por mi entorno de desarrollo estándar, casi siempre digo «IPython más un editor de texto». Normalmente escribo un programa, lo pruebo una y otra vez y depuro cada fragmento en notebooks de IPython o Jupyter. También resulta útil poder jugar con los datos de forma interactiva y verificar visualmente que un determinado conjunto de manipulaciones de datos está haciendo lo que tiene que hacer. Librerías como `pandas` y `NumPy` están diseñadas para que resulte productivo utilizarlas en el shell.

Pero cuando se trata de crear software, quizá algunos usuarios prefieren emplear un IDE (*Integrated Development Environment*: entorno de desarrollo integrado) que

disponga de más funciones, en lugar de un editor como Emacs o Vim, que ofrecen directamente un entorno mínimo. Estos son algunos editores que puede explorar:

- PyDev (gratuito), un IDE integrado en la plataforma Eclipse.
- PyCharm de JetBrains (con suscripción para usuarios comerciales, gratuito para desarrolladores de código abierto).
- Python Tools for Visual Studio (para usuarios de Windows).
- Spyder (gratuito), un IDE incluido actualmente con Anaconda.
- Komodo IDE (comercial).

Debido a la popularidad de Python, la mayoría de los editores de texto, como VS Code y Sublime Text 2, ofrecen un excelente soporte de Python.

1.5 Comunidad y conferencias

Aparte de las búsquedas en Internet, las distintas listas de correo de Python científicas y asociadas a datos suelen ser útiles y proporcionan respuestas. Algunas de ellas, por echarles un vistazo, son las siguientes:

- pydata: Una lista de grupos de Google para cuestiones relacionadas con Python para análisis de datos y pandas.
- pystatsmodels: para preguntas relacionadas con statsmodels o pandas.
- Lista de correo generalmente para scikit-learn (scikit-learn@python.org) y aprendizaje automático en Python.
- numpy-discussion: para cuestiones relacionadas con NumPy.
- scipy-user: para preguntas generales sobre SciPy o Python científico.

No he incluido deliberadamente URL para estas listas de correo en caso de que cambien. Se pueden encontrar fácilmente buscando en Internet.

Todos los años tienen lugar muchas conferencias en todo el mundo para programadores de Python. Si le gustaría conectar con otros programadores de Python que compartan sus intereses, le animo a que explore asistiendo a una, si es posible. Muchas conferencias disponen de ayudas económicas para quienes no pueden permitirse pagar la entrada o el viaje a la conferencia. Algunas a tener en cuenta son:

- PyCon y EuroPython: las dos conferencias principales de Python en Norteamérica y Europa, respectivamente.
- SciPy y EuroSciPy: conferencias orientadas a la computación científica en Norteamérica y Europa, respectivamente.
- PyData: una serie de conferencias regionales a nivel mundial destinadas a la ciencia de datos y a casos de uso en análisis de datos.

- Conferencias PyCon internacionales y regionales (consulte <https://pycon.org> si desea una lista completa).

1.6 Navegar por este libro

Si nunca había programado antes en Python, quizá le convenga estudiar a fondo los capítulos 2 y 3, en los que he incluido un condensado tutorial sobre las funciones del lenguaje Python y los notebooks del shell de IPython y de Jupyter. Todo ello supone conocimientos previos necesarios para continuar con el resto del libro. Si ya tiene experiencia con Python, quizá prefiera saltarse estos capítulos.

A continuación ofrezco una breve introducción a las funciones esenciales de NumPy, dejando el uso más avanzado de NumPy para el apéndice A. Luego presento pandas y dedico el resto del libro a temas de análisis de datos relacionados con pandas, NumPy y matplotlib (para visualización). He estructurado el material de un modo incremental, aunque en ocasiones haya referencias menores entre capítulos, pues hay casos en los que se utilizan conceptos que todavía no han sido introducidos.

Aunque los lectores puedan tener muchos objetivos distintos para su trabajo, las tareas requeridas suelen entrar dentro de una serie de amplios grupos determinados:

- Interactuar con el mundo exterior: Leer y escribir con distintos formatos de archivos y almacenes de datos.
- Preparación: Limpieza, procesado, combinación, normalización, remodelado, segmentación y transformación de datos para su análisis.
- Transformación: Aplicar operaciones matemáticas y estadísticas a grupos de conjuntos de datos para obtener de ellos nuevos conjuntos de datos (por ejemplo, agregando una tabla grande por variables de grupo).
- Modelado y computación: Conectar los datos con modelos estadísticos, algoritmos de aprendizaje automático u otras herramientas computacionales.
- Presentación: Crear visualizaciones interactivas, gráficos estáticos o resúmenes de texto.

Códigos de ejemplo

La mayoría de los códigos de ejemplo del libro se muestran con entrada y salida, como si aparecieran ejecutados en el shell de IPython o en notebooks de Jupyter:

```
In [5]: EJEMPLO DE CÓDIGO
Out[5]: SALIDA
```

Quando vea un código como este, la intención es que lo escriba en el bloque In de su entorno de codificación y lo ejecute pulsando la tecla **Intro** (o **Mayús-Intro** en Jupyter).

Tendría que ver un resultado similar al que se muestra en el bloque Out.

He cambiado la configuración predeterminada de la salida de la consola en NumPy y pandas para mejorar la legibilidad y brevedad a lo largo del libro. Por ejemplo, quizá vea más dígitos de precisión impresos en datos numéricos. Para lograr el resultado exacto que aparece en el libro, puede ejecutar el siguiente código de Python antes de ponerse con los ejemplos:

```
import numpy as np
import pandas as pd
pd.options.display.max_columns = 20
pd.options.display.max_rows = 20
pd.options.display.max_colwidth = 80
np.set_printoptions(precision=4, suppress=True)
```

Datos para los ejemplos

Los conjuntos de datos para los ejemplos de cada capítulo están guardados en un repositorio GitHub (<https://github.com/wesm/pydata-book>) o en un duplicado en Gitee (<https://gitee.com/wesmckinn/pydata-book>), si no es posible acceder a GitHub. Se pueden descargar utilizando el sistema de control de versiones Git de la línea de comandos o descargando un archivo zip del repositorio ubicado en el sitio web. Si tiene problemas, entre en el sitio web del libro original (<https://wesmckinney.com/book>) para obtener instrucciones actualizadas sobre cómo conseguir los materiales del libro.

Si descarga un archivo zip que contiene los conjuntos de datos de ejemplo, deberá entonces extraer por completo el contenido de dicho archivo en un directorio y acceder finalmente a él desde el terminal antes de proceder a la ejecución de los ejemplos de código del libro:

```
$ pwd
/home/wesm/book-materials
```

```
$ ls
appa.ipynb      ch05.ipynb      ch09.ipynb      ch13.ipynb      README.md
ch02.ipynb      ch06.ipynb      ch10.ipynb      COPYING          requirements.txt
ch03.ipynb      ch07.ipynb      ch11.ipynb      datasets
ch04.ipynb      ch08.ipynb      ch12.ipynb      examples
```

He hecho todo lo que estaba en mi mano para asegurar que el repositorio GitHub contiene todo lo necesario para reproducir los ejemplos, pero quizá haya cometido errores

u omisiones. Si es así, le pido por favor que me envíe un email a: book@wesmckinney.com.

La mejor manera de informar de errores hallados en el libro es consultando la página de erratas del libro original en el sitio web de O'Reilly (<https://www.oreilly.com/catalog/errata.csp?isbn=0636920519829>).

Convenios de importación

La comunidad Python ha adoptado distintos convenios de nomenclatura para los módulos más utilizados:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

import statsmodels as sm
```

Esto significa que cuando vea `np.arange`, se trata de una referencia a la función `arange` de NumPy. Esto es así porque se considera mala praxis en desarrollo de software de Python importarlo todo (`from numpy import *`) de un paquete de gran tamaño como NumPy.

Fundamentos del lenguaje Python, IPython y Jupyter Notebooks

Cuando escribí la primera edición de este libro en 2011 y 2012, existían menos recursos para aprender a realizar análisis de datos en Python. Esto era en parte un problema del tipo «huevo y gallina»; muchas librerías que ahora damos por sentadas, como *pandas*, *scikit-learn* y *statsmodels*, eran entonces inmaduras comparándolas con lo que son ahora. En la actualidad existe cada vez más literatura sobre ciencia de datos, análisis de datos y aprendizaje automático, que complementa los trabajos previos sobre computación científica genérica destinados a científicos computacionales, físicos y profesionales de otros campos de investigación. También hay libros excelentes para aprender el lenguaje de programación Python y convertirse en un ingeniero de software eficaz.

Como este libro está destinado a ser un texto introductorio para el trabajo con datos en Python, me parece valioso disponer de una visión de conjunto de algunas de las funciones más importantes de las estructuras integradas en Python y de sus librerías desde el punto de vista de la manipulación de datos. Por esta razón, solo presentaré en este capítulo y el siguiente la información suficiente para que sea posible seguir el resto del libro.

Buena parte de este libro se centra en analítica de tablas y herramientas de preparación de datos para trabajar con conjuntos de datos lo bastante reducidos como para poder manejarse en un ordenador personal. Para utilizar estas herramientas, en ocasiones es necesario hacer ciertas modificaciones para organizar datos desordenados en un formato tabular (o estructurado) más sencillo de manejar. Por suerte, Python es un lenguaje ideal para esto. Cuanto mayor sea la capacidad de manejo por parte del usuario del lenguaje Python y sus tipos de datos integrados, más fácil será preparar nuevos conjuntos de datos para su análisis.

Algunas de las herramientas de este libro se exploran mejor desde una sesión activa de IPython o Jupyter. En cuanto aprenda a iniciar IPython y Jupyter, le recomiendo que siga los ejemplos, de modo que pueda experimentar y probar distintas cosas. Al igual que con un entorno de consola con teclado, desarrollar una buena memoria recordando los comandos habituales también forma parte de la curva de aprendizaje.



Hay conceptos introductorios de Python que este capítulo no trata, como, por ejemplo, las clases y la programación orientada a objetos, que quizá encuentre útil en su incursión en el análisis de datos con Python.

Para intensificar sus conocimientos del lenguaje Python, le recomiendo que complemente este capítulo con el tutorial oficial de Python (<https://docs.python.org>) y posiblemente con uno de los muchos libros de calidad que existen sobre programación genérica con Python. Algunas recomendaciones para empezar son las siguientes:

- *Python Cookbook*, tercera edición, de David Beazley y Brian K. Jones (O'Reilly).
- *Fluent Python*, de Luciano Ramalho (O'Reilly).
- *Effective Python*, de Brett Slatkin (Addison-Wesley).

2.1 El intérprete de Python

Python es un lenguaje interpretado. El intérprete de Python pone en marcha un programa que ejecuta una sentencia cada vez. El intérprete interactivo estándar de Python puede activarse desde la

línea de comandos con el comando python:

```
$ python
Python 3.10.4 | packaged by conda-forge | (main, Mar 24 2022, 17:38:57)
[GCC 10.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 5
>>> print(a)
5
```

El símbolo >>> que se puede ver en el código es el *prompt* en el que se escriben las expresiones y fragmentos de código. Para salir del intérprete de Python, se puede escribir **exit()** o pulsar **Control-D** (únicamente en Linux y macOS).

Ejecutar programas de Python es tan sencillo como llamar a python con un archivo .py como primer argumento. Supongamos que habíamos creado hello_world.py con este contenido:

```
print("Hello world")
```

Se puede ejecutar utilizando el siguiente comando (el archivo hello_world.py debe estar en su directorio de trabajo actual del terminal):

```
$ python hello_world.py
Hello world
```

Mientras algunos programadores ejecutan su código Python de esta forma, los que realizan análisis de datos o ciencia computacional emplean IPython, un intérprete de Python mejorado, o bien notebooks de Jupyter, cuadernos de código basados en la web y creados inicialmente dentro del proyecto IPython. Ofreceré en este capítulo una introducción al uso de IPython y Jupyter, y en el apéndice A profundizaré más en la funcionalidad de IPython. Al utilizar el comando %run, IPython ejecuta el código del archivo especificado dentro del mismo proceso, y permite así explorar los resultados de forma interactiva cuando ha terminado:

```
$ ipython
Python 3.10.4 | packaged by conda-forge | (main, Mar 24 2022, 17:38:57)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.31.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: %run hello_world.py
Hello world

In [2]:
```

El *prompt* predeterminado de IPython adopta el estilo numerado In [2]:, a diferencia del *prompt* estándar >>>.

2.2 Fundamentos de IPython

En esta sección nos pondremos en marcha con el shell de IPython y el notebook de Jupyter, y presentaré algunos de los conceptos básicos.

Ejecutar el shell de IPython

Se puede lanzar el shell de IPython en la línea de comandos exactamente igual que se lanza el intérprete de Python, excepto que hay que usar el comando `ipython`:

```
$ ipython
Python 3.10.4 | packaged by conda-forge | (main, Mar 24 2022, 17:38:57)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.31.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: a = 5

In [2]: a
Out[2]: 5
```

Es posible ejecutar sentencias arbitrarias de Python escribiéndolas y pulsando **Return** (o **Intro**). Al escribir solo una variable en IPython, devuelve una representación de cadena de texto del objeto:

```
In [5]: import numpy as np

In [6]: data = [np.random.standard_normal() for i in range(7)]

In [7]: data
Out[7]:
[-0.20470765948471295,
 0.47894333805754824,
-0.5194387150567381,
-0.55573030434749,
 1.9657805725027142,
 1.3934058329729904,
 0.09290787674371767]
```

Las dos primeras líneas son sentencias de código Python; la segunda sentencia crea una variable llamada `data` que se refiere a un diccionario Python de reciente creación. La última línea imprime el valor de `data` en la consola.

Muchos tipos de objetos Python están formateados para que sean más legibles, o queden mejor al imprimirlos, que es distinto de la impresión normal que se consigue con `print`. Si se imprimiera la variable `data` anterior en el intérprete de Python estándar, sería mucho menos legible:

```
>>> import numpy as np
>>> data = [np.random.standard_normal() for i in range(7)]
>>> print(data)
>>> data
[-0.5767699931966723, -0.1010317773535111, -1.7841005313329152,
-1.524392126408841, 0.22191374220117385, -1.9835710588082562,
-1.6081963964963528]
```

IPython ofrece además formas de ejecutar bloques arbitrarios de código (mediante una especie de método copiar y pegar con pretensiones) y fragmentos enteros de código Python. Se puede utilizar el notebook de Jupyter para trabajar con bloques de código más grandes, como veremos muy pronto.

Ejecutar el notebook de Jupyter

Uno de los componentes principales del proyecto Jupyter es el notebook, un tipo de documento interactivo para código, texto (incluyendo Markdown), visualizaciones de datos y otros resultados. El

notebook de Jupyter interactúa con los *kernels*, que son implementaciones del protocolo de computación interactivo de Jupyter específicos para distintos lenguajes de programación. El *kernel* de Python Jupyter emplea el sistema IPython para su comportamiento subyacente. Para iniciar Jupyter, ejecute el comando `jupyter notebook` en un terminal:

```
$ jupyter notebook
[I 15:20:52.739 NotebookApp] Serving notebooks from local directory:
/home/wesm/code/pydata-book
[I 15:20:52.739 NotebookApp] 0 active kernels
[I 15:20:52.739 NotebookApp] The Jupyter Notebook is running at:
http://localhost:8888/?token=0a77b52fefe52ab83e3c35dff8de121e4bb443a63f2d...
[I 15:20:52.740 NotebookApp] Use Control-C to stop this server and shut down
all kernels (twice to skip confirmation).
Created new window in existing browser session.
To access the notebook, open this file in a browser:
```

```
file:///home/wesm/.local/share/jupyter/runtime/nbserver-185259-open.html
```

Or copy and paste one of these URLs:

```
http://localhost:8888/?token=0a77b52fefe52ab83e3c35dff8de121e4...
```

```
or http://127.0.0.1:8888/?token=0a77b52fefe52ab83e3c35dff8de121e4...
```

En muchas plataformas, Jupyter se abrirá automáticamente en el navegador web predeterminado (a menos que se inicie con el parámetro `--no-browser`). De otro modo, se puede acceder a la dirección HTTP que aparece al iniciar el notebook, en este caso <http://localhost:8888/?token=0a77b52fefe52ab83e3c35dff8de121e4bb443a63f2d3055>. En la figura 2.1 se muestra cómo se ve esto en Google Chrome.



Muchas personas utilizan Jupyter como entorno local, pero también se puede desplegar en servidores y se puede acceder a él remotamente. No daré aquí más detalles al respecto, pero le animo a que explore este tema en Internet si le resulta relevante para sus necesidades.

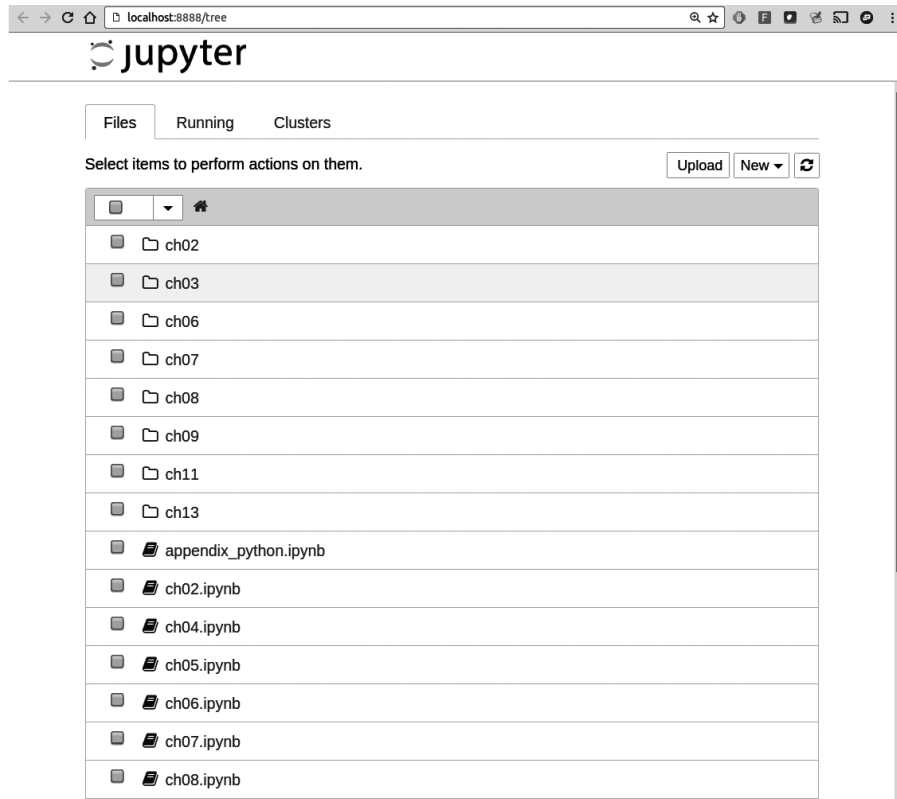


Figura 2.1. Página de inicio de Jupyter notebook.

Para crear un nuevo notebook, haga clic en el botón **New** (nuevo) y seleccione la opción **Python** 3. Debería verse algo parecido a lo que muestra la figura 2.2. Si es su primera vez, pruebe a hacer clic en la celda vacía y escriba una línea de código Python. A continuación, pulse **Mayús-Intro** para ejecutarlo.

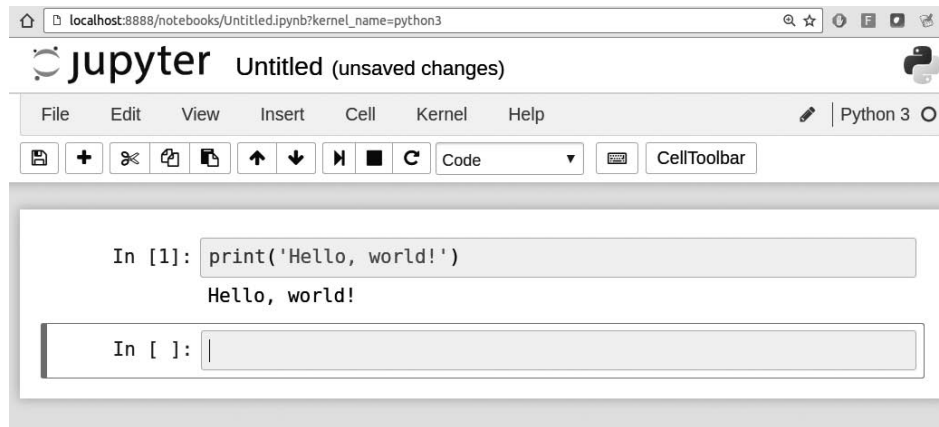


Figura 2.2. Vista de un nuevo notebook de Jupyter.

Al guardar el notebook utilizando **Save and Checkpoint** (guardar y comprobar) en el menú **File** (archivo), se crea un archivo con la extensión `.ipynb`. Se trata de un formato de archivo autónomo

que incluye todo lo que contiene en ese momento el notebook (incluyendo el resultado de código ya evaluado). Estos archivos pueden ser abiertos y editados por otros usuarios de Jupyter.

Para renombrar un notebook abierto, haga clic en su título en la parte superior de la página y escriba el nuevo, pulsando **Enter** al terminar.

Para abrir un notebook existente, ponga el archivo en el mismo directorio en el que inició el proceso del notebook (o en una subcarpeta contenida en él) y después haga clic en el nombre desde la página de inicio. Puede probar con los notebooks de mi repositorio wesm/pydata-book de GitHub; consulte además la figura 2.3.

Cuando quiera cerrar un notebook, haga clic en el menú **File** (archivo) y elija **Close and Halt** (cerrar y detener). Si solamente cierra la pestaña del navegador, el proceso de Python asociado al notebook se mantendrá en funcionamiento en segundo plano.

Aunque el notebook de Jupyter pueda parecer una experiencia distinta al shell de IPython, casi todos los comandos y herramientas de este capítulo se pueden utilizar en ambos entornos.

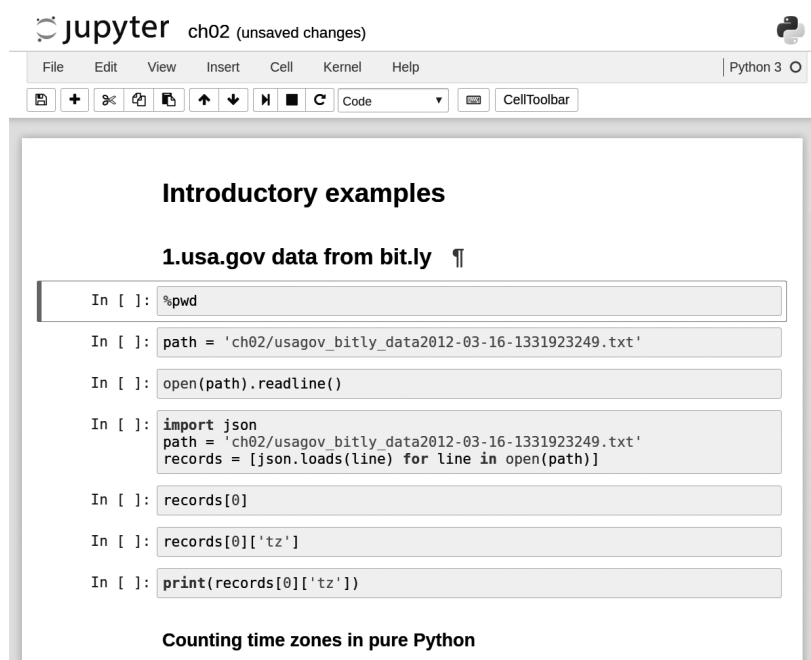


Figura 2.3. Vista de ejemplo de Jupyter de un notebook existente.

Autocompletado

A primera vista, el shell de IPython parece una versión en apariencia distinta al intérprete de Python estándar (que se abre con python). Una de las principales mejoras con respecto al shell de Python estándar es el autocompletado, que puede encontrarse en muchos IDE u otros entornos de análisis computacional interactivos. Mientras se escriben expresiones en el shell, pulsar la tecla **Tab** buscará en el espacio de nombres cualquier variable (objetos, funciones, etc.) que coincida con los caracteres que se han escrito hasta ahora y mostrará los resultados en un cómodo menú desplegable:

```
In [1]: an_apple = 27
```

```
In [2]: an_example = 42
```

```
In [3]: an<Tab>
an_apple an_example any
```

En este ejemplo se puede observar que IPython mostró las dos variables que definí, además de la función integrada `any`. Además, es posible completar métodos y atributos de cualquier objeto tras escribir un punto:

```
In [3]: b = [1, 2, 3]
```

```
In [4]: b.<Tab>
```

<code>append()</code>	<code>count()</code>	<code>insert()</code>	<code>reverse()</code>
<code>clear()</code>	<code>extend()</code>	<code>pop()</code>	<code>sort()</code>
<code>copy()</code>	<code>index()</code>	<code>remove()</code>	

Lo mismo aplica a los módulos:

```
In [1]: import datetime
```

```
In [2]: datetime.<Tab>
```

<code>date</code>	<code>MAXYEAR</code>	<code>timedelta</code>
<code>datetime</code>	<code>MINYEAR</code>	<code>timezone</code>
<code>datetime_CAPI</code>	<code>time</code>	<code>tzinfo</code>



Hay que tener en cuenta que IPython oculta por defecto métodos y atributos que empiezan por el carácter de subrayado, como por ejemplo métodos mágicos y métodos y atributos «privados» internos, para evitar desordenar la pantalla (y confundir a los usuarios principiantes). Estos también se pueden autocompletar, pero primero hay que escribir un subrayado para verlos. Si prefiere ver siempre estos métodos en el autocompletado, puede cambiar la opción en la configuración de IPython. Consulte la documentación de IPython (<https://ipython.readthedocs.io/en/stable/>) para averiguar cómo hacerlo.

El autocompletado funciona en muchos contextos, aparte de la búsqueda interactiva en el espacio de nombres y de completar atributos de objeto o módulo. Al escribir cualquier cosa que parezca la ruta de un archivo (incluso en una cadena de texto de Python), pulsar la tecla **Tab** completará cualquier cosa en el sistema de archivos de su ordenador que coincida con lo que haya escrito.

Combinada con el comando `%run` (consulte la sección «El comando `%run`», del apéndice B), esta funcionalidad puede ahorrar muchas pulsaciones de teclas.

Otro área en el que el autocompletado ahorra tiempo es al escribir argumentos de palabra clave de funciones, que incluso incluyen el signo `=` (véase la figura 2.4).

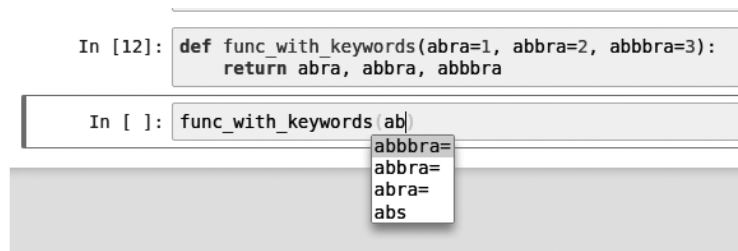


Figura 2.4. Autocompletar palabras clave de función en un notebook de Jupyter.

Le echaremos un vistazo más detallado a las funciones en un momento.

Introspección

Utilizar un signo de interrogación (?) antes o después de una variable mostrará información general sobre el objeto:

```
In [1]: b = [1, 2, 3]
```

```
In [2]: b?
Type: list
String form: [1, 2, 3]
Length: 3
Docstring:
Built-in mutable sequence.
```

If no argument is given, the constructor creates a new empty list.
The argument must be an iterable if specified.

```
In [3]: print?
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file: a file-like object (stream); defaults to the current sys.stdout.
sep: string inserted between values, default a space.
end: string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
Type: builtin_function_or_method

Esto se denomina introspección de objetos. Si el objeto es una función o un método de instancia, la cadena de documentación o *docstring*, si se ha definido, también se mostrará. Supongamos que habíamos escrito la siguiente función (que se puede reproducir en IPython o Jupyter):

```
def add_numbers(a, b):
    """
    Add two numbers together
    Returns
    ____
    the_sum : type of arguments
    """
    return a + b
```

Utilizando entonces ? aparece la *docstring*:

```
In [6]: add_numbers?
Signature: add_numbers(a, b)
Docstring:
Add two numbers together
Returns
____
the_sum : type of arguments
File: <ipython-input-9-6a548a216e27>
Type: function
```

El signo de interrogación tiene un último uso, destinado a buscar en el espacio de nombres de IPython de una manera similar a la línea de comandos estándar de Unix o Windows. Una serie de caracteres combinados con el carácter comodín (*) mostrará todos los nombres que coinciden con la expresión comodín. Por ejemplo, podríamos obtener una lista de todas las funciones del espacio de nombres de alto nivel de NumPy que contengan load:

```
In [9]: import numpy as np
```

```
In [10]: np.*load*?
np.__loader__
np.load
np.loads
np.loadtxt
```

2.3 Fundamentos del lenguaje Python

En esta sección ofreceré un resumen de los conceptos esenciales de programación de Python y de la mecánica del lenguaje. En el siguiente capítulo entraré en más detalle sobre estructuras de datos, funciones y otras herramientas internas de Python.

Semántica del lenguaje

El diseño del lenguaje Python se distingue por su énfasis en la legibilidad, simplicidad y claridad. Algunas personas llegan a compararlo con un «pseudocódigo ejecutable».

Sangrado, no llaves

Python emplea espacios en blanco (tabuladores o espacios) para estructurar el código, en lugar de utilizar llaves, como en muchos otros lenguajes como R, C++, Java y Perl. Veamos un bucle for de un algoritmo de ordenación:

```
for x in array:
    if x < pivot:
        less.append(x)
    else:
        greater.append(x)
```

El signo de los dos puntos denota el comienzo de un bloque de código sangrado, tras el cual todo el código debe estar sangrado en la misma cantidad hasta el final del bloque.

Les guste o no, el espacio en blanco con significado es una realidad en la vida de los programadores de Python. Aunque pueda parecer raro al principio, es de esperar que con el tiempo uno se acostumbre.



Recomiendo enérgicamente reemplazar el tabulador por cuatro espacios como sangrado predeterminado. Muchos editores de texto incluyen un parámetro en su configuración que reemplaza los tabuladores por espacios de manera automática (¡actívelo!). Los notebooks de IPython y Jupyter insertarán automáticamente cuatro espacios en líneas nuevas después de dos puntos y sustituirán también los tabuladores por cuatro espacios.

Como ha podido ver hasta ahora, las sentencias de Python no tienen que terminar tampoco por punto y coma. No obstante, el punto y coma se puede utilizar para separar varias sentencias que están en una sola línea:

```
a = 5; b = 6; c = 7
```

Normalmente no se ponen varias sentencias en una sola línea en Python, porque puede hacer que el código sea menos legible.

Todo es un objeto

Una característica importante del lenguaje Python es la consistencia de su modelo de objetos. Cada número, cadena de texto, estructura de datos, función, clase, módulo, etc. existe en el intérprete de Python en su propia «caja», lo que se denomina objeto Python. Cada objeto tiene un tipo asociado (por ejemplo, entero, texto o función) y unos datos internos. En la práctica esto consigue que el lenguaje sea muy flexible, pues hasta las funciones pueden ser tratadas como un objeto más.

Comentarios

Cualquier texto precedido por el carácter de la almohadilla # es ignorado por el intérprete de Python. A menudo se emplea para añadir comentarios al código. En ocasiones quizá interese también excluir determinados bloques de código sin borrarlos. Una solución es convertir esos bloques en comentarios:

```
results = []
for line in file_handle:
    # deja las líneas vacías por ahora
    # if len(line) == 0:
    # continúa
    results.append(line.replace("foo", "bar"))
```

Los comentarios pueden aparecer también tras una línea de código ejecutado. Aunque algunos programadores prefieren colocar los comentarios en la línea que precede a una determinada línea de código, en ocasiones puede resultar útil:

```
print("Reached this line")           # Sencillo informe de estado
```

Llamadas a funciones y a métodos de objeto

Se llama a las funciones utilizando paréntesis y pasándoles cero o más argumentos, asignando de manera opcional el valor devuelto a una variable:

```
result = f(x, y, z)
g()
```

Casi todos los objetos de Python tienen funciones asociadas, conocidas como métodos, que tienen acceso al contenido interno del objeto. Se les puede llamar utilizando esta sintaxis:

```
obj.some_method(x, y, z)
```


Las funciones pueden admitir argumentos posicionales y de palabra clave:

```
result = f(a, b, c, d=5, e="foo")
```

Veremos esto con más detalle más adelante.

Pasar o asignar variables y argumentos

Al asignar una variable (o nombre) en Python, se está creando una referencia al objeto que aparece al lado derecho del signo igual. En términos prácticos, supongamos una lista de enteros:

```
In [8]: a = [1, 2, 3]
```

Imaginemos que asignamos a a una nueva variable b:

```
In [9]: b = a
```

```
In [10]: b  
Out[10]: [1, 2, 3]
```

En algunos lenguajes, la asignación de b hará que se copien los datos [1, 2, 3]. En Python, a y b se refieren ahora en realidad al mismo objeto, la lista original [1, 2, 3] (véase una representación de esto en la figura 2.5). Puede probarlo por sí mismo añadiendo un elemento a a y después examinando b:

```
In [11]: a.append(4)
```

```
In [12]: b  
Out[12]: [1, 2, 3, 4]
```

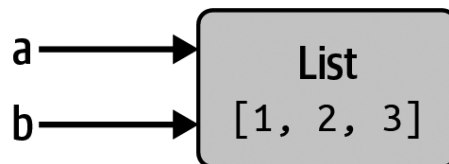


Figura 2.5. Dos referencias al mismo objeto.

Comprender la semántica de las referencias de Python y cuándo, cómo y por qué se copian los datos es especialmente importante cuando se trabaja con conjuntos de datos grandes en Python.



La asignación también se denomina vinculación, porque estamos asociando un nombre a un objeto. A los nombres de variables que han sido asignados se les llama, asimismo, variables vinculadas.

Cuando se pasan objetos como argumentos a una función, se crean nuevas variables locales que hacen referencia a los objetos originales sin copiar nada. Si se vincula un nuevo objeto a una variable dentro de una función, no se sobrescribirá una variable del mismo nombre que esté en el «ámbito» (o *scope*) exterior de la función (el «ámbito padre»). Por eso es posible modificar el interior de un argumento mutable. Supongamos que tenemos la siguiente función:

```
In [13]: def append_element(some_list, element):
```

```
.....:         some_list.append(element)
```

Entonces tenemos:

```
In [14]: data = [1, 2, 3]
In [15]: append_element(data, 4)
In [16]: data
Out[16]: [1, 2, 3, 4]
```

Referencias dinámicas, tipos fuertes

Las variables en Python no tienen un tipo inherente asociado; una variable puede hacer referencia a un tipo distinto de objeto simplemente haciendo una asignación. No hay problema con lo siguiente:

```
In [17]: a = 5
In [18]: type(a)
Out[18]: int
In [19]: a = "foo"
In [20]: type(a)
Out[20]: str
```

Las variables son nombres para objetos dentro de un espacio de nombres determinado; la información del tipo se almacena en el propio objeto. Algunos observadores podrían concluir apresuradamente que Python no es un «lenguaje de tipos». Pero esto no es cierto; veamos este ejemplo:

```
In [21]: "5" + 5
_____
TypeError                                Traceback (most recent call last)
<ipython-input-21-7fe5aa79f268> in <module>
--> 1 "5" + 5
TypeError: can only concatenate str (not "int") to str
```

En algunos lenguajes, la cadena de texto "5" se podría convertir de manera implícita en un entero, produciendo así 10. En otros lenguajes, el entero 5 podría transformarse en una cadena de texto, produciendo así el texto concatenado "55". En Python, estas transformaciones implícitas no están permitidas.

A este respecto, decimos que Python es un lenguaje de tipos fuertes, lo que significa que cada objeto tiene un tipo (o clase) específico, y las conversiones implícitas solo se producen en determinadas circunstancias permitidas, como las siguientes:

```
In [22]: a = 4.5
In [23]: b = 2

# Formateado de cadena de texto, lo veremos después
In [24]: print(f"a is {type(a)}, b is {type(b)}")
a is <class 'float'>, b is <class 'int'>
```

```
In [25]: a / b
Out[25]: 2.25
```

Aquí, incluso aunque b sea un entero, se convierte implícitamente en un tipo float para la operación de división.

Conocer el tipo de un objeto es importante, y útil para poder escribir funciones que puedan manejar muchos tipos distintos de entradas. Se puede comprobar que un objeto es una instancia de un determinado tipo utilizando la función `isinstance`:

```
In [26]: a = 5

In [27]: isinstance(a, int)
Out[27]: True
```

`isinstance` puede aceptar una tupla de tipos si queremos comprobar que el tipo de un objeto está entre los presentes en la tupla:

```
In [28]: a = 5; b = 4.5

In [29]: isinstance(a, (int, float))
Out[29]: True

In [30]: isinstance(b, (int, float))
Out[30]: True
```

Atributos y métodos

Los objetos en Python suelen tener tanto atributos (otros objetos Python almacenados «dentro» del objeto) como métodos (funciones asociadas a un objeto que pueden tener acceso a sus datos internos). A ambos se accede mediante la sintaxis `obj.attribute_name`:

```
In [1]: a = "foo"

In [2]: a.<Pulse Tab>
```

<code>capitalize()</code>	<code>index()</code>	<code>isspace()</code>	<code>removesuffix()</code>	<code>startswith()</code>
<code>casefold()</code>	<code>isprintable()</code>	<code>istitle()</code>	<code>replace()</code>	<code>strip()</code>
<code>center()</code>	<code>isalnum()</code>	<code>isupper()</code>	<code>rfind()</code>	<code>swapcase()</code>
<code>count()</code>	<code>isalpha()</code>	<code>join()</code>	<code>rindex()</code>	<code>title()</code>
<code>encode()</code>	<code>isascii()</code>	<code>ljust()</code>	<code>rjust()</code>	<code>translate()</code>
<code>endswith()</code>	<code>isdecimal()</code>	<code>lower()</code>	<code>rpartition()</code>	
<code>expandtabs()</code>	<code>isdigit()</code>	<code>lstrip()</code>	<code>rsplit()</code>	
<code>find()</code>	<code>isidentifier()</code>	<code>maketrans()</code>	<code>rstrip()</code>	
<code>format()</code>	<code>islower()</code>	<code>partition()</code>	<code>split()</code>	
<code>format_map()</code>	<code>isnumeric()</code>	<code>removeprefix()</code>	<code>splitlines()</code>	

A los atributos y métodos también se puede acceder mediante la función `getattr`:

```
In [32]: getattr(a, "split")
Out[32]: <function str.split(sep=None, maxsplit=-1)>
```

Aunque no utilizaremos mucho las funciones `getattr` y las relacionadas `hasattr` y `setattr` en este libro, se pueden emplear de una forma muy eficaz para escribir código genérico reutilizable.

Duck typing

Es posible que con frecuencia no importe el tipo de un objeto, sino solamente si incluye determinados métodos o tiene un cierto comportamiento. A esto se le denomina «*duck typing*», que se podría traducir como «tipado de pato», aunque no tiene mucho sentido, porque se emplea esta expresión por el dicho «si camina como un pato y grazna como un pato, entonces es un pato». Por ejemplo, es posible verificar que un objeto es iterable si implementa el protocolo iterador. Para muchos objetos, esto significa que tiene un «método mágico» `__iter__`, aunque una alternativa y mejor forma de comprobarlo es intentar usar la función `iter`:

```
In [33]:          def isiterable(obj):
.....:            try:
.....:              iter(obj)
.....:              return True
.....:            except TypeError: # no iterable
.....:              return False
```

Esta función devolvería `True` para cadenas de texto, así como para la mayoría de los tipos de colección de Python:

```
In [34]:          isiterable("a string")
Out[34]:          True
In [35]:          isiterable([1, 2, 3])
Out[35]:          True
In [36]:          isiterable(5)
Out[36]:          False
```

Importaciones

En Python, un módulo es simplemente un archivo con la extensión `.py` que contiene código Python. Supongamos que tenemos el siguiente módulo:

```
# some_module.py
PI = 3.14159
def f(x):
    return x + 2
def g(a, b):
    return a + b
```

Si queremos acceder a las variables y funciones definidas en `some_module.py`, desde otro archivo del mismo directorio podemos hacer esto:

```
import some_module
result = some_module.f(5)

pi = some_module.PI
```

O como alternativa:

```
from some_module import g, PI
result = g(5, PI)
```

Utilizando la palabra clave `as` se les puede asignar a las importaciones distintos nombres de variable:

```
import some_module as sm
from some_module import PI as pi, g as gf

r1 = sm.f(pi)
r2 = gf(6, pi)
```

Operadores binarios y comparaciones

La mayor parte de las operaciones matemáticas binarias y las comparaciones utilizan la misma sintaxis matemática ya conocida empleada en otros lenguajes de programación:

```
In [37]:          5 - 7
Out[37]:          -2
In [38]:          12 + 21.5
Out[38]:          33.5
In [39]:          5 <= 2
Out[39]:          False
```

Véanse en la tabla 2.1 todos los operadores binarios disponibles.

Tabla 2.1. Operadores binarios.

Operación	Descripción
$a + b$	Suma a y b
$a - b$	Resta b de a
$a * b$	Multiplifica a por b
a / b	Divide a por b
$a // b$	División de piso de a por b , es decir, calcula el cociente de la división entre a y b , sin tener en cuenta el resto
$a ** b$	Eleva a a la potencia de b
$a \& b$	True si tanto a como b son True; para enteros, toma AND <i>bitwise</i> (o a nivel de bit)
$a b$	True si a o b son True; para enteros, toma OR <i>bitwise</i>
$a ^ b$	Para booleanos, True si a o b es True, pero no ambos; para enteros, toma OR EXCLUSIVO <i>bitwise</i>
$a == b$	True si a es igual a b
$a != b$	True si a no es igual a b
$a < b$, $a <= b$	True si a es menor (menor o igual) que b
$a > b$, $a >= b$	True si a es mayor (mayor o igual) que b

Operación	Descripción
<code>a is b</code>	True si a y b hacen referencia al mismo objeto Python
<code>a is not b</code>	True si a y b hacen referencia a distintos objetos Python

Para comprobar si dos variables se refieren al mismo objeto, utilizamos la palabra clave `is`, que no se puede usar para verificar que dos objetos no sean el mismo:

```
In [40]: a = [1, 2, 3]
In [41]: b = a
In [42]: c = list(a)
In [43]: a is b
Out[43]: True
In [44]: a is not c
Out[44]: True
```

Como la función `list` siempre crea una lista nueva de Python (por ejemplo, una copia), podemos estar seguros de que `c` es distinto de `a`. Comparar con `is` no es lo mismo que utilizar el operador `==`, porque en este caso tenemos:

```
In [45]: a == c
Out[45]: True
```

Habitualmente se utilizan `is` e `is not` también para comprobar que una variable sea `None`, ya que solamente hay una instancia de `None`:

```
In [46]: a = None
In [47]: a is None
Out[47]: True
```

Objetos mutables e inmutables

Muchos objetos en Python, como listas, diccionarios, arrays NumPy y la mayoría de los tipos (clases) definidos por el usuario, son mutables. Esto significa que el objeto o los valores que contiene se pueden modificar:

```
In [48]: a_list = ["foo", 2, [4, 5]]
In [49]: a_list[2] = (3, 4)
In [50]: a_list
Out[50]: ['foo', 2, (3, 4)]
```

Otros, como cadenas de texto y tuplas, son inmutables, lo que significa que sus datos internos no pueden cambiarse:

```
In [51]: a_tuple = (3, 5, (4, 5))
In [52]: a_tuple[1] = "four"
```

```

TypeError                                Traceback (most recent call last)
<ipython-input-52-cd2a018a7529> in <module>
--> 1 a_tuple[1] = "four"
TypeError: 'tuple' object does not support item assignment

```

Conviene recordar que, simplemente porque el hecho de que se pueda mutar un objeto, no significa que siempre se deba hacer. Estas acciones se conocen como efectos colaterales. Por ejemplo, al escribir una función, cualquier efecto colateral debería ser explícitamente comunicado al usuario en la documentación de la función o en los comentarios. Si es posible, recomiendo tratar de evitar efectos colaterales y favorecer la inmutabilidad, incluso aunque pueda haber objetos mutables implicados.

Tipos escalares

Python tiene un pequeño conjunto de tipos integrados para manejar datos numéricos, cadenas de texto, valores booleanos (`True` o `False`) y fechas y horas. A estos tipos de “valores sencillos” se les denomina tipos escalares; en este libro nos referiremos a ellos simplemente como escalares. Consulte en la tabla 2.2 una lista de los principales tipos escalares. El manejo de fechas y horas se tratará de manera individual, porque estos valores son suministrados por el módulo `datetime` de la librería estándar.

Tabla 2.2. Tipos escalares estándares de Python.

Tipo	Descripción
<code>None</code>	El valor «null» de Python (solo existe una instancia del objeto <code>None</code>)
<code>str</code>	Tipo cadena de texto; contiene textos Unicode
<code>bytes</code>	Datos binarios sin procesar
<code>float</code>	Número de punto flotante de precisión doble (observe que no existe un tipo <code>double</code> distinto)
<code>bool</code>	Un valor booleano <code>True</code> o <code>False</code>
<code>int</code>	Entero de precisión arbitraria

Tipos numéricos

Los principales tipos de Python para los números son `int` y `float`. Un `int` puede almacenar números arbitrariamente grandes:

```

In [53]: ival = 17239871

In [54]: ival ** 6
Out[54]: 26254519291092456596965462913230729701102721

```

Los números de punto flotante se representan con el tipo `float` de Python. Internamente, cada uno es un valor de precisión doble. También se pueden expresar con notación científica:

```

In [55]: fval = 7.243

```

```
In [56]: fval2 = 6.78e-5
```

La división de enteros que no dé como resultado otro número entero siempre producirá un número de punto flotante:

```
In [57]: 3 / 2
Out[57]: 1.5
```

Para lograr una división de enteros al estilo de C (que no tiene en cuenta el resto si el resultado no es un número entero), utilizamos el operador de división de piso `//`:

```
In [58]: 3 // 2
Out[58]: 1
```

Cadenas de texto

Mucha gente utiliza Python por sus capacidades internas de manejo de cadenas de texto. Se pueden escribir literales de cadena empleando o bien comillas simples `'` o dobles `"` (en general se utilizan más las dobles comillas):

```
a = 'one way of writing a string'
b = "another way"
```

El tipo cadena de texto de Python es `str`.

Para cadenas de texto de varias líneas con saltos de línea, se pueden utilizar tres comillas, sencillas `'''` o dobles `"""`:

```
c = """
This is a longer string that
spans multiples lines
"""
```

Quizá sorprenda el hecho de que esta cadena de texto `c` contenga realmente cuatro líneas de texto; los saltos de línea después de `"""` y después de `lines` están incluidos. Podemos contar los caracteres de la nueva línea con el método `count` sobre `c`:

```
In [60]: c.count("\n")
Out[60]: 3
```

Las cadenas de texto de Python son inmutables; no se pueden modificar:

```
In [61]: a = "this is a string"
In [62]: a[10] = "f"
```

```
TypeError                                Traceback (most recent call last)
<ipython-input-62-3b2d95f10db4> in <module>
--> 1 a[10] = "f"
TypeError: 'str' object does not support item assignment
```

Para interpretar este mensaje de error, léalo de abajo a arriba. Hemos intentado reemplazar el carácter («*item*») de la posición 10 por la letra `"f"`, pero esto no está permitido para objetos de cadena

de texto. Si necesitamos modificar una cadena de texto, tenemos que utilizar una función o un método que cree una nueva cadena, como el método `replace` para cadenas de texto:

```
In [63]: b = a.replace("string", "longer string")
```

```
In [64]: b
Out[64]: 'this is a longer string'
```

Tras esta operación, la variable `a` no ha sido modificada:

```
In [65]: a
Out[65]: 'this is a string'
```

Muchos objetos de Python se pueden transformar en una cadena de texto utilizando la función `str`:

```
In [66]: a = 5.6
In [67]: s = str(a)
In [68]: print(s)
5.6
```

Las cadenas de texto son una secuencia de caracteres Unicode y, por lo tanto, se les puede tratar igual que otras secuencias, como por ejemplo las listas y las tuplas:

```
In [69]: s = "python"
In [70]: list(s)
Out[70]: ['p', 'y', 't', 'h', 'o', 'n']

In [71]: s[:3]
Out[71]: 'pyt'
```

La sintaxis `s[:3]` se denomina *slicing* y se implementa para muchos tipos de secuencias de Python. Explicaremos esto con más detalle más adelante, pues se utiliza mucho en este libro.

El carácter de la barra invertida `\` es un carácter de escape, lo que significa que se emplea para especificar caracteres especiales, como el carácter de línea nueva `\n`, o caracteres Unicode. Para escribir un literal de cadena con barras invertidas, es necesario escaparlos:

```
In [72]: s = "12\\34"
In [73]: print(s)
12\34
```

Si tenemos una cadena de texto con muchas barras invertidas y ningún carácter especial, podría ser bastante molesto. Por suerte se puede poner delante de la primera comilla del texto una `r`, que significa que los caracteres se deben interpretar tal y como están:

```
In [74]: s = r"this\has\nospecial\characters"
In [75]: s
Out[75]: 'this\\has\\no\\special\\characters'
```

La `r` significa *raw* (sin procesar).

Sumar dos cadenas las concatena y produce una nueva:

```
In [76]: a = "this is the first half "
```

```
In [77]: b = "and this is the second half"
```

```
In [78]: a + b
```

```
Out[78]: 'this is the first half and this is the second half'
```

La creación de plantillas o formato de cadenas de texto es otro tema importante. Con la llegada de Python 3, esto puede hacerse de más formas que antes, así que aquí describiremos brevemente la mecánica de uno de los interfaces principales. Los objetos de cadena de texto tienen un método `format` que se puede utilizar para sustituir argumentos formateados dentro de la cadena, produciendo una nueva:

```
In [79]: template = "{0:.2f} {1:s} are worth US${2:d}"
```

En esta cadena:

- `{0:.2f}` significa formatear el primer argumento como un número de punto flotante con dos decimales.
- `{1:s}` significa formatear el segundo argumento como una cadena de texto.
- `{2:d}` significa formatear el tercer argumento como un entero exacto.

Para sustituir los argumentos para estos parámetros de formato, pasamos una secuencia de argumentos al método `format`:

```
In [80]: template.format(88.46, "Argentine Pesos", 1)
```

```
Out[80]: '88.46 Argentine Pesos are worth US$1'
```

Python 3.6 introdujo una nueva característica llamada «cadenas-f» (abreviatura de «literales de cadena formateados») que puede lograr que sea aún más cómoda la creación de cadenas formateadas.

Para crear una cadena-f, escribimos el carácter `f` justo antes de un literal de cadena. Dentro de la propia cadena de texto, encerramos las expresiones Python en llaves para sustituir el valor de la expresión por la cadena formateada:

```
In [81]: amount = 10
```

```
In [82]: rate = 88.46
```

```
In [83]: currency = "Pesos"
```

```
In [84]: result = f"{amount} {currency} is worth US${amount / rate}"
```

Pueden añadirse especificadores de formato tras cada expresión, utilizando la misma sintaxis que hemos visto antes con las plantillas de cadena:

```
In [85]: f"{amount} {currency} is worth US${amount / rate:.2f}"
```

```
Out[85]: '10 Pesos is worth US$0.11'
```

El formato de cadenas de texto es un tema de gran profundidad; existen varios métodos y distintas opciones y modificaciones disponibles para controlar cómo se formatean los valores en la cadena

resultante.



Para saber más, le recomiendo que consulte la documentación oficial de Python (<https://docs.python.org/3/library/string.html>).

Bytes y Unicode

En el Python moderno (es decir, Python 3.0 y superior), Unicode se ha convertido en el tipo de cadena de texto de primer orden en permitir un manejo más sólido de texto ASCII y no ASCII. En versiones más antiguas de Python, las cadenas de texto eran todo bytes sin una codificación Unicode explícita. Se podía convertir a Unicode suponiendo que se conociera la codificación del carácter. Aquí muestro una cadena de texto Unicode de ejemplo con caracteres no ASCII:

```
In [86]: val = "español"
```

```
In [87]: val
Out[87]: 'español'
```

Podemos convertir esta cadena Unicode en su representación de bytes UTF-8 utilizando el método `encode`:

```
In [88]: val_utf8 = val.encode("utf-8")
```

```
In [89]: val_utf8
Out[89]: b'espa\xc3\xb1ol'
```

```
In [90]: type(val_utf8)
Out[90]: bytes
```

Suponiendo que conocemos la codificación Unicode de un objeto bytes, podemos volver atrás utilizando el método `decode`:

```
In [91]: val_utf8.decode("utf-8")
Out[91]: 'español'
```

Aunque ahora se prefiere utilizar UTF-8 para cualquier codificación, por razones históricas es posible encontrar datos en diferentes y variadas codificaciones:

```
In [92]: val.encode("latin1")
Out[92]: b'espa\xf1ol'
```

```
In [93]: val.encode("utf-16")
Out[93]: b'\xff\xfe\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'
```

```
In [94]: val.encode("utf-16le")
Out[94]: b'\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'
```

Es más habitual encontrar objetos bytes cuando se trabaja con archivos, donde quizá no sea deseable decodificar implícitamente todos los datos a cadenas Unicode.

Booleanos

Los dos valores booleanos de Python se escriben como True y False. Las comparaciones y otras expresiones condicionales evalúan a True o False. Los valores booleanos se combinan con las palabras clave and y or:

```
In [95]: True and True
Out[95]: True

In [96]: False or True
Out[96]: True
```

Cuando se convierten a números, False se convierte en 0 y True en 1:

```
In [97]: int(False)
Out[97]: 0

In [98]: int(True)
Out[98]: 1
```

La palabra clave not invierte un valor booleano de True a False o viceversa:

```
In [99]: a = True

In [100]: b = False

In [101]: not a
Out[101]: False

In [102]: not b
Out[102]: True
```

Conversión de tipos (*Type casting*)

Los tipos str, bool, int y float son también funciones que se pueden usar para convertir valores a dichos tipos:

```
In [103]: s = "3.14159"

In [104]: fval = float(s)

In [105]: type(fval)
Out[105]: float

In [106]: int(fval)
Out[106]: 3

In [107]: bool(fval)
Out[107]: True

In [108]: bool(0)
Out[108]: False
```

Observe que la mayoría de los valores que no son cero, cuando se convierten a bool, son True.

None

None es el tipo de valor de Python nulo o *null*.

```
In [109]: a = None
```

```
In [110]: a is None
Out[110]: True
```

```
In [111]: b = 5
```

```
In [112]: b is not None
Out[112]: True
```

None es también un valor predeterminado habitual para argumentos de función:

```
def add_and_maybe_multiply(a, b, c=None):
    result = a + b
    if c is not None:
        result = result * c
    return result
```

Fechas y horas

El módulo `datetime` integrado de Python ofrece los tipos `datetime`, `date` y `time`. El tipo `datetime` combina la información almacenada en `date` y `time` y es el más utilizado:

```
In [113]: from datetime import datetime, date, time
```

```
In [114]: dt = datetime(2011, 10, 29, 20, 30, 21)
```

```
In [115]: dt.day
Out[115]: 29
```

```
In [116]: dt.minute
Out[116]: 30
```

Dada una instancia `datetime`, se pueden extraer los objetos equivalentes `date` y `time` llamando a métodos de la `datetime` del mismo nombre:

```
In [117]: dt.date()
Out[117]: datetime.date(2011, 10, 29)
```

```
In [118]: dt.time()
Out[118]: datetime.time(20, 30, 21)
```

El método `strftime` formatea un `datetime` como cadena de texto:

```
In [119]: dt.strftime("%Y-%m-%d %H:%M")
Out[119]: '2011-10-29 20:30'
```

Las cadenas de texto se pueden convertir (analizar) en objetos `datetime` con la función `strptime`:

```
In [120]: datetime.strptime("20091031", "%Y%m%d")
Out[120]: datetime.datetime(2009, 10, 31, 0, 0)
```

En la tabla 11.2 se puede consultar la lista completa de especificaciones de formato.

Cuando estamos agregando o agrupando de otro modo datos de series temporales, de vez en cuando es útil reemplazar campos de hora de una serie de datetimes (por ejemplo, sustituyendo los campos minute y second por cero):

```
In [121]: dt_hour = dt.replace(minute=0, second=0)
```

```
In [122]: dt_hour
```

```
Out[122]: datetime.datetime(2011, 10, 29, 20, 0)
```

Como datetime.datetime es un tipo inmutable, métodos como este siempre producen nuevos objetos. Así, en el código de arriba, dt no es modificado por replace:

```
In [123]: dt
```

```
Out[123]: datetime.datetime(2011, 10, 29, 20, 30, 21)
```

La diferencia de dos objetos datetime produce un tipo datetime.timedelta:

```
In [124]: dt2 = datetime.datetime(2011, 11, 15, 22, 30)
```

```
In [125]: delta = dt2 - dt
```

```
In [126]: delta
```

```
Out[126]: datetime.timedelta(days=17, seconds=7179)
```

```
In [127]: type(delta)
```

```
Out[127]: datetime.timedelta
```

El resultado timedelta(17, 7179) indica que el timedelta codifica un desplazamiento de 17 días y 7,179 segundos.

Sumar un timedelta a un datetime produce un nuevo datetime movido de su sitio:

```
In [128]: dt
```

```
Out[128]: datetime.datetime(2011, 10, 29, 20, 30, 21)
```

```
In [129]: dt + delta
```

```
Out[129]: datetime.datetime(2011, 11, 15, 22, 30)
```

Control de flujo

Python tiene varias palabras clave internas para lógica condicional, bucles y otros conceptos estándares de control de flujo, que pueden encontrarse en otros lenguajes de programación.

if, elif y else

La sentencia if es uno de los tipos de sentencia de control de flujo más conocidos. Comprueba una condición que, si es True, evalúa el código del bloque que le sigue:

```
x = -5
if x < 0:
    print("It's negative")
```

Una sentencia `if` puede ir seguida de manera opcional por uno o más bloques `elif` y un bloque multifuncional `else` si todas las condiciones son `False`:

```
if x < 0:
    print("It's negative")
elif x == 0:
    print("Equal to zero")
elif 0 < x < 5:
    print("Positive but smaller than 5")
else:
    print("Positive and larger than or equal to 5")
```

Si alguna de las condiciones es `True`, no se alcanzará ningún bloque `elif` o `else`. Con una condición compuesta utilizando `and` o `or`, las condiciones se evalúan de izquierda a derecha y se produce un cortocircuito:

```
In [130]: a = 5; b = 7
In [131]: c = 8; d = 4

In [132]: if a < b or c > d:
.....:         print("Made it")
Made it
```

En este ejemplo, la comparación `c > d` nunca resulta evaluada porque la primera comparación era `True`.

También es posible encadenar comparaciones:

```
In [133]: 4 > 3 > 2 > 1
Out[133]: True
```

Bucles for

Los bucles `for` se utilizan para aplicar determinadas instrucciones a una colección (como una lista o una tupla) o a un iterador. La sintaxis estándar de un bucle `for` es:

```
for value in collection:
    # hace algo con value
```

Se puede avanzar un bucle `for` a la siguiente repetición, saltando el resto del bloque, mediante la palabra clave `continue`. Veamos este código, que suma enteros de una lista y omite valores `None`:

```
sequence = [1, 2, None, 4, None, 5]
total = 0
for value in sequence:
    if value is None:
        continue
    total += value
```

Un bucle `for` puede darse por terminado también con la palabra clave `break`. Este código suma elementos de una lista hasta que se llega a un 5:

```
sequence = [1, 2, 0, 4, 6, 5, 2, 1]
total_until_5 = 0
for value in sequence:
    if value == 5:
        break
    total_until_5 += value
```

La palabra clave `break` solo finaliza el bucle `for` más interno; los bucles `for` exteriores seguirán ejecutándose:

```
In [134]:          for i in range(4):
.....:          for j in range(4):
.....:              if j > i:
.....:                  break
.....:              print((i, j))
.....:
(0, 0)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
(2, 2)
(3, 0)
(3, 1)
(3, 2)
(3, 3)
```

Como veremos con más detalle, si los elementos de la colección o iterador son secuencias (tuplas o listas, por ejemplo), se pueden desempaquetar cómodamente en variables de la sentencia del bucle `for`:

```
for a, b, c in iterator:
    # hace algo
```

Bucles while

Un bucle `while` especifica una condición y un bloque de código que se va ejecutar hasta que la condición evalúe a `False` o el bucle sea finalizado explícitamente con `break`:

```
x = 256
total = 0
while x > 0:
    if total > 500:
        break
    total += x
    x = x // 2
```

`pass`

pass es la sentencia «no operativa» (es decir, que no hace nada) de Python. Se puede utilizar en los bloques en los que no hay que realizar ninguna acción (o como marcador para código que aún no se ha implementado); solo es necesario porque Python emplea el espacio en blanco para delimitar los bloques:

```
if x < 0:
    print("negative!")
elif x == 0:
    # TODO: pone algo inteligente aquí
    pass
else:
    print("positive!")
```

range

La función range genera una secuencia de enteros espaciados por igual:

```
In [135]: range(10)
Out[135]: range(0, 10)

In [136]: list(range(10))
Out[136]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Se puede dar un inicio, un final y un paso o incremento (que puede ser negativo):

```
In [137]: list(range(0, 20, 2))
Out[137]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

In [138]: list(range(5, 0, -1))
Out[138]: [5, 4, 3, 2, 1]
```

Como se puede comprobar, range produce enteros hasta el valor final pero sin incluirlo; se suele utilizar para aplicar instrucciones a distintas secuencias según un índice:

```
In [139]: seq = [1, 2, 3, 4]

In [140]:
.....:         for i in range(len(seq)):
.....:             print(f"element {i}: {seq[i]}")
element 0:         1
element 1:         2
element 2:         3
element 3:         4
```

Aunque se pueden utilizar funciones como list para almacenar todos los enteros generados por range en alguna otra estructura de datos, a menudo la forma de iteración predeterminada será la que el usuario decida. El siguiente fragmento de código suma todos los números de 0 a 99.999 que son múltiplos de 3 o 5:

```
In [141]:         total = 0
In [142]:         for i in range(100_000):
.....:             # % es el operador módulo
.....:             if i % 3 == 0 or i % 5 == 0:
```

```
.....:          total += i
In [143]:          print(total)
2333316668
```

Aunque el rango generado puede ser arbitrariamente grande, el uso de la memoria en cualquier momento determinado puede ser muy pequeño.

2.4 Conclusión

Este capítulo ha ofrecido una breve introducción a algunos conceptos básicos del lenguaje Python y a los entornos de programación IPython y Jupyter. En el siguiente capítulo trataremos muchos tipos de datos y funciones integradas y utilidades de entrada-salida que se emplearán continuamente en todo el libro.

Estructuras de datos integrados, funciones y archivos

Este capítulo aborda ciertas capacidades del lenguaje Python que emplearemos profusamente a lo largo del libro. Librerías adicionales, como `pandas` y `NumPy`, añaden funcionalidad computacional avanzada para grandes conjuntos de datos, pero están diseñadas para usarse junto con las herramientas de manipulación de datos integradas en Python.

Empezaremos con las estructuras de datos esenciales de Python: tuplas, listas, diccionarios y conjuntos; después hablaremos de la creación de nuestras propias funciones de Python reutilizables y, por último, veremos la mecánica de los objetos archivo de Python y su interacción con el disco duro local del usuario.

3.1 Estructuras de datos y secuencias

Las estructuras de datos de Python son sencillas, a la vez que potentes. Dominar su uso es fundamental para convertirse en un programador competente de Python. Empezamos por los tipos de secuencia más utilizados, es decir, las tuplas, las listas y los diccionarios.

Tupla

Una tupla es una secuencia de objetos Python inmutable y de longitud fija que, una vez asignada, no puede modificarse. La forma más sencilla de crear una tupla es mediante una secuencia de valores separados por comas y encerrados entre paréntesis:

```
In [2]: tup = (4, 5, 6)
```

```
In [3]: tup
```

```
Out[3]: (4, 5, 6)
```

En muchos contextos, los paréntesis se pueden omitir, de modo que también podríamos haber escrito:

```
In [4]: tup = 4, 5, 6
```

```
In [5]: tup
```

```
Out[5]: (4, 5, 6)
```

Es posible convertir cualquier secuencia o iterador en una tupla invocando `tuple`:

```
In [6]: tuple([4, 0, 2])
```

```
Out[6]: (4, 0, 2)
```

```
In [7]: tup = tuple('string')
```

```
In [8]: tup
```

```
Out[8]: ('s', 't', 'r', 'i', 'n', 'g')
```

Se puede acceder a los elementos mediante los paréntesis cuadrados [], como con casi todos los tipos de secuencia. Como en C, C++, Java y muchos otros lenguajes, en Python las secuencias están indexadas al cero:

```
In [9]: tup[0]
```

```
Out[9]: 's'
```

Cuando se definen tuplas con expresiones más complicadas, a menudo es necesario encerrar los valores entre paréntesis, como en este ejemplo de creación de una tupla de tuplas:

```
In [10]: nested_tup = (4, 5, 6), (7, 8)
```

```
In [11]: nested_tup
```

```
Out[11]: ((4, 5, 6), (7, 8))
```

```
In [12]: nested_tup[0]
```

```
Out[12]: (4, 5, 6)
```

```
In [13]: nested_tup[1]
```

```
Out[13]: (7, 8)
```

Aunque los objetos almacenados en una tupla pueden ser mutables por sí mismos, una vez la tupla se ha creado, ya no es posible cambiar qué objeto está almacenado en cada espacio:

```
In [14]: tup = tuple(['foo', [1, 2], True])
In [15]: tup[2] = False
```

```
TypeError          Traceback (most recent call last)
<ipython-input-15-b89d0c4ae599> in <module>
--> 1 tup[2] = False
TypeError: 'tuple' object does not support item assignment
```

Si un objeto contenido en una tupla es mutable, como una lista por ejemplo, se puede modificar en su ubicación:

```
In [16]: tup[1].append(3)

In [17]: tup
Out[17]: ('foo', [1, 2, 3], True)
```

Es posible concatenar tuplas, produciendo tuplas más largas, con el operador +:

```
In [18]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[18]: (4, None, 'foo', 6, 0, 'bar')
```

Si se multiplica una tupla por un entero, como con las listas, se logra concatenar tantas copias de la tupla como el resultado de la multiplicación:

```
In [19]: ('foo', 'bar') * 4
Out[19]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

Hay que tener en cuenta que los objetos como tales no se copian, solo las referencias a ellos.

Desempaquetar tuplas

Si se intenta asignar una expresión de variables de estilo tupla, Python intentará desempaquetar el valor situado a la derecha del signo igual:

```
In [20]: tup = (4, 5, 6)
```

```
In [21]: a, b, c = tup
```

```
In [22]: b
```

```
Out[22]: 5
```

Incluso las secuencias con tuplas anidadas se pueden desempaquetar:

```
In [23]: tup = 4, 5, (6, 7)
```

```
In [24]: a, b, (c, d) = tup
```

```
In [25]: d
```

```
Out[25]: 7
```

Utilizando esta funcionalidad se pueden intercambiar fácilmente nombres de variables, una tarea que en muchos lenguajes podría ser algo así:

```
tmp = a
```

```
a = b
```

```
b = tmp
```

Pero, en Python, el intercambio puede realizarse de este modo:

```
In [26]: a, b = 1, 2
```

```
In [27]: a
```

```
Out[27]: 1
```

```
In [28]: b
```

```
Out[28]: 2
```

```
In [29]: b, a = a, b
```

```
In [30]: a
```

```
Out[30]: 2
```

```
In [31]: b
```

```
Out[31]: 1
```

El desempaquetado de variables se suele utilizar para iterar sobre secuencias de tuplas o listas:

```
In [32]:      seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
In [33]:      for a, b, c in seq:
.....:         print(f'a={a}, b={b}, c={c}')
a=1, b=2, c=3
a=4, b=5, c=6
a=7, b=8, c=9
```

También se emplea para devolver varios valores de una función. Hablaremos de esto con más detalle posteriormente.

Hay situaciones en las que podría ser interesante «retirar» varios elementos del inicio de una tupla. Existe una sintaxis especial que puede hacerlo, `*rest`, que se emplea también en firmas de función para capturar una lista arbitrariamente larga de argumentos posicionales:

```
In [34]: values = 1, 2, 3, 4, 5
```

```
In [35]: a, b, *rest = values
```

```
In [36]: a
Out[36]: 1
```

```
In [37]: b
Out[37]: 2
```

```
In [38]: rest
Out[38]: [3, 4, 5]
```

En ocasiones, el término `rest` es algo que se puede querer desechar; el nombre `rest` no tiene nada de especial. Por una cuestión de costumbre, muchos programadores de Python emplean el carácter de subrayado (`_`) para variables no deseadas:

```
In [39]: a, b, *_ = values
```

Métodos de tupla

Como el tamaño y contenido de una tupla no se puede modificar, tiene poca incidencia en los métodos de instancia. Uno especialmente útil (disponible también en las listas) es `count`, que cuenta el número de apariciones de un valor:

```
In [40]: a = (1, 2, 2, 2, 3, 4, 2)
```

```
In [41]: a.count(2)
```

```
Out[41]: 4
```

Listas

A diferencia de las tuplas, las listas tienen longitud variable y su contenido se puede modificar. Las listas son mutables, y pueden definirse utilizando paréntesis cuadrados `[]` o la función de tipo `list`:

```
In [42]: a_list = [2, 3, 7, None]
```

```
In [43]: tup = ("foo", "bar", "baz")
```

```
In [44]: b_list = list(tup)
```

```
In [45]: b_list
```

```
Out[45]: ['foo', 'bar', 'baz']
```

```
In [46]: b_list[1] = "peekaboo"
```

```
In [47]: b_list
```

```
Out[47]: ['foo', 'peekaboo', 'baz']
```

Las listas y las tuplas son semánticamente similares (aunque las tuplas no se pueden modificar) y ambas se pueden emplear en muchas funciones de manera intercambiable.

La función integrada `list` se usa con frecuencia en proceso de datos como una forma de materializar una expresión iteradora o generadora:

```
In [48]: gen = range(10)
```

```
In [49]: gen
```

```
Out[49]: range(0, 10)
```

```
In [50]: list(gen)
```



```
Out[50]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Añadir y eliminar elementos

Es posible añadir elementos al final de la lista con el método `append`:

```
In [51]: b_list.append("dwarf")
```

```
In [52]: b_list
```

```
Out[52]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

Utilizando `insert` se puede insertar un elemento en una determinada posición de la lista:

```
In [53]: b_list.insert(1, "red")
```

```
In [54]: b_list
```

```
Out[54]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

El índice de inserción debe estar entre 0 y la longitud de la lista, inclusive.



`insert` es computacionalmente caro comparado con `append`, porque las referencias a posteriores elementos tienen que moverse internamente para dejar sitio al nuevo elemento. Si es necesario insertar elementos tanto al principio como al final de una secuencia, se puede hacer uso de `collections.deque`, una cola de doble extremo optimizada con este fin y que se puede encontrar en la librería estándar de Python.

La operación inversa a `insert` es `pop`, que elimina y devuelve un elemento en un determinado índice:

```
In [55]: b_list.pop(2)
```

```
Out[55]: 'peekaboo'
```

```
In [56]: b_list
```

```
Out[56]: ['foo', 'red', 'baz', 'dwarf']
```

Se pueden eliminar elementos por valor con `remove`, que localiza el primero de los valores y lo elimina de la lista:

```
In [57]: b_list.append("foo")
```

```
In [58]: b_list
Out[58]: ['foo', 'red', 'baz', 'dwarf', 'foo']

In [59]: b_list.remove("foo")

In [60]: b_list
Out[60]: ['red', 'baz', 'dwarf', 'foo']
```

Si el rendimiento no es un problema, empleando `append` y `remove` es posible usar una lista de Python como una estructura de datos de estilo conjunto (aunque Python ya tiene objetos de conjunto, que trataremos más adelante).

Podemos comprobar que una lista contiene un valor mediante la palabra clave `in`:

```
In [61]: "dwarf" in b_list
Out[61]: True
```

Puede emplearse la palabra clave `not` para negar a `in`:

```
In [62]: "dwarf" not in b_list
Out[62]: False
```

Verificar que una lista contenga un valor es mucho más lento que hacerlo con diccionarios y conjuntos (que presentaremos en breve), pues Python realiza una exploración lineal por los valores de la lista, mientras que puede revisar los otros (basados en tablas *hash*) en todo momento.

Concatenar y combinar listas

Igual que con las tuplas, sumar dos listas con `+` las concatena:

```
In [63]: [4, None, "foo"] + [7, 8, (2, 3)]
Out[63]: [4, None, 'foo', 7, 8, (2, 3)]
```

Si ya tenemos una lista definida, se le pueden añadir varios elementos utilizando el método `extend`:

```
In [64]: x = [4, None, "foo"]
```

```
In [65]: x.extend([7, 8, (2, 3)])
```

```
In [66]: x
```

```
Out[66]: [4, None, 'foo', 7, 8, (2, 3)]
```

La concatenación de listas mediante suma es una operación comparativamente cara, porque se tiene que crear una nueva lista y copiar en ella los objetos. Normalmente es preferible usar `extend` para agregar elementos a una lista ya existente, especialmente si se está formando una lista grande. Por lo tanto:

```
everything = []
for chunk in list_of_lists:
    everything.extend(chunk)
```

Es más rápido que la alternativa concatenante:

```
everything = []
for chunk in list_of_lists:
    everything = everything + chunk
```

Ordenar

Es posible ordenar una lista en el momento (sin crear un objeto nuevo) llamando a su función `sort`:

```
In [67]: a = [7, 2, 5, 1, 3]
```

```
In [68]: a.sort()
```

```
In [69]: a
```

```
Out[69]: [1, 2, 3, 5, 7]
```

`sort` tiene varias opciones que de vez en cuando resultan útiles. Una de ellas es la capacidad para pasar una clave de ordenación secundaria (es decir, una función que produce un valor que se utiliza para ordenar los objetos). Por ejemplo, podríamos ordenar una colección de cadenas de texto por sus longitudes:

```
In [70]: b = ["saw", "small", "He", "foxes", "six"]
```

```
In [71]: b.sort(key=len)
```

```
In [72]: b
```

```
Out[72]: ['He', 'saw', 'six', 'small', 'foxes']
```

Pronto veremos la función `sorted`, que puede producir una copia ordenada de una secuencia general.

Corte o rebanado

Se pueden seleccionar partes de la mayoría de los tipos de secuencia empleando la notación de corte, que en su forma básica consiste en pasar `start:stop` al operador de indexado `[]`:

```
In [73]: seq = [7, 2, 3, 7, 5, 6, 0, 1]
```

```
In [74]: seq[1:5]
```

```
Out[74]: [2, 3, 7, 5]
```

Los cortes o rebanadas (*slices*) también se pueden asignar con una secuencia:

```
In [75]: seq[3:5] = [6, 3]
```

```
In [76]: seq
```

```
Out[76]: [7, 2, 3, 6, 3, 6, 0, 1]
```

Mientras el elemento del índice `start` está incluido, el índice `stop` no lo está, de modo que el número de elementos del resultado es `stop-start`.

Se puede omitir el `start` o el `stop`, en cuyo caso su valor predeterminado es el inicio de la secuencia y el final de la secuencia, respectivamente:

```
In [77]: seq[:5]
```

```
Out[77]: [7, 2, 3, 6, 3]
```

```
In [78]: seq[3:]
```

```
Out[78]: [6, 3, 6, 0, 1]
```

Los índices negativos cortan la secuencia relativa al final:

```
In [79]: seq[-4:]  
Out[79]: [3, 6, 0, 1]
```

```
In [80]: seq[-6:-2]  
Out[80]: [3, 6, 3, 6]
```

Acostumbrarse a la semántica del corte cuesta un poco, especialmente si procedemos de R o MATLAB. Véase en la figura 3.1 una útil ilustración del corte con enteros positivos y negativos. En la figura, los índices se muestran en los «extremos», para así mostrar fácilmente dónde se inician y detienen las selecciones de corte utilizando índices positivos o negativos.

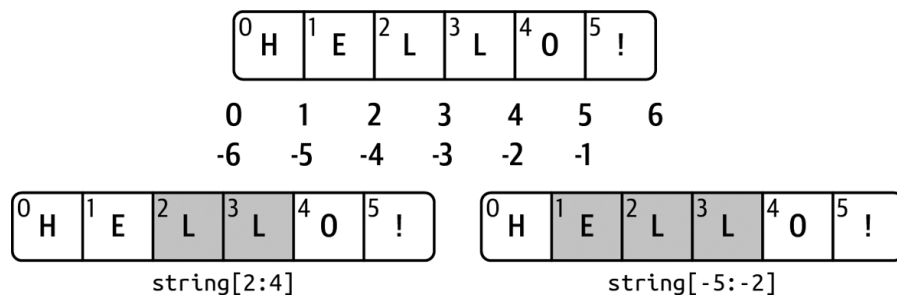


Figura 3.1. Ilustración de los convenios de corte de Python.

Se puede utilizar también un step después de un segundo signo de dos puntos para, por ejemplo, tomar los elementos que quedan:

```
In [81]: seq[::2]  
Out[81]: [7, 3, 3, 0]
```

Un uso inteligente de esto es pasar -1, que tiene el útil efecto de revertir una lista o tupla:

```
In [82]: seq[::-1]  
Out[82]: [1, 0, 6, 3, 6, 3, 2, 7]
```

Diccionario

Puede que el diccionario o dict sea la estructura de datos integrada de Python más importante de todas. En otros lenguajes de programación, a los

diccionarios también se les llama mapas *hash* o arrays asociativos. Un diccionario almacena una colección de pares clave-valor, donde la clave y el valor son objetos Python. Cada clave está asociada a un valor, de modo que dicho valor se pueda recuperar, insertar, modificar o borrar convenientemente dada una determinada clave. Una forma de crear un diccionario es utilizando llaves {} y signos de dos puntos para separar claves y valores:

```
In [83]: empty_dict = {}
```

```
In [84]: d1 = {"a": "some value", "b": [1, 2, 3, 4]}
```

```
In [85]: d1
```

```
Out[85]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

Se puede acceder a los elementos, insertarlos o formar conjuntos con ellos utilizando la misma sintaxis que para acceder a los elementos de una lista o tupla:

```
In [86]: d1[7] = "an integer"
```

```
In [87]: d1
```

```
Out[87]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

```
In [88]: d1["b"]
```

```
Out[88]: [1, 2, 3, 4]
```

Es posible comprobar que un diccionario contiene una clave empleando la misma sintaxis usada para verificar si una lista o tupla contiene un valor:

```
In [89]: "b" in d1
```

```
Out[89]: True
```

Se pueden borrar valores mediante la palabra clave `del` o bien con el método `pop` (que devuelve simultáneamente el valor y borra la clave):

```
In [90]: d1[5] = "some value"
```

```
In [91]: d1
```

```
Out[91]:
```

```
{'a'          : 'some value',  
'b'          : [1, 2, 3, 4],  
7:           'an integer',  
5:           'some value'}
```

```
In [92]: d1["dummy"] = "another value"
```

```
In [93]: d1  
Out[93]:  
{'a'          : 'some value',  
'b'          : [1, 2, 3, 4],  
7:           'an integer',  
5:           'some value',  
'dummy': 'another value'}
```

```
In [94]: del d1[5]
```

```
In [95]: d1  
Out[95]:  
{'a'          : 'some value',  
'b'          : [1, 2, 3, 4],  
7:           'an integer',  
'dummy': 'another value'}
```

```
In [96]: ret = d1.pop("dummy")
```

```
In [97]: ret  
Out[97]: 'another value'
```

```
In [98]: d1  
Out[98]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an  
integer'}
```

El método `keys` y `values` proporciona iteradores de las claves y los valores del diccionario, respectivamente. El orden de las claves depende del orden de su inserción, y estas funciones dan como resultado las claves y los valores en el mismo orden respectivo:

```
In [99]: list(d1.keys())  
Out[99]: ['a', 'b', 7]
```

```
In [100]: list(d1.values())
Out[100]: ['some value', [1, 2, 3, 4], 'an integer']
```

Si es necesario iterar por las claves y los valores, se puede utilizar el método `items` para hacer lo propio sobre las mismas como tuplas de dos:

```
In [101]: list(d1.items())
Out[101]: [('a', 'some value'), ('b', [1, 2, 3, 4]), (7, 'an integer')]
```

Se puede combinar un diccionario con otro con el método `update`:

```
In [102]: d1.update({"b": "foo", "c": 12})

In [103]: d1
Out[103]: {'a': 'some value', 'b': 'foo', 7: 'an integer', 'c': 12}
```

El método `update` cambia los diccionarios en el momento, de modo que cualquier clave existente en los datos pasados a `update` hará que se descarten sus anteriores valores.

Crear diccionarios a partir de secuencias

Es habitual terminar a veces con dos secuencias que se desean emparejar elemento a elemento en un diccionario. Como primer paso, se podría escribir un fragmento de código como este:

```
mapping = {}
for key, value in zip(key_list, value_list):
    mapping[key] = value
```

Como un diccionario es básicamente una colección de tuplas de dos, la función `dict` acepta una lista de tuplas de dos o pares:

```
In [104]: tuples = zip(range(5), reversed(range(5)))

In [105]: tuples
Out[105]: <zip at 0x7fefe4553a00>
```



```
In [106]: mapping = dict(tuples)
```

```
In [107]: mapping
```

```
Out[107]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

Después hablaremos de las comprensiones de diccionarios, otra forma de construir diccionarios.

Valores predeterminados

Es común tener lógica como la siguiente:

```
if key in some_dict:
    value = some_dict[key]
else:
    value = default_value
```

Así, los métodos de diccionario `get` y `pop` pueden tomar un valor predeterminado que devolverán, de forma que el bloque anterior `if-else` se podría escribir con más sencillez así:

```
value = some_dict.get(key, default_value)
```

De forma predeterminada, `get` devolverá `None` si la clave no está presente, mientras que `pop` producirá una excepción. Con valores de configuración, puede ser que los valores de un diccionario sean otro tipo de colección, como una lista. Por ejemplo, podríamos pensar en categorizar una lista de palabras por sus letras iniciales como un diccionario de listas:

```
In [108]: words = ["apple", "bat", "bar", "atom", "book"]
```

```
In [109]: by_letter = {}
```

```
In [110]: for word in words:
.....:     letter = word[0]
.....:     if letter not in by_letter:
.....:         by_letter[letter] = [word]
.....:     else:
.....:         by_letter[letter].append(word)
```

```
.....:
.....:
```

```
In [111]: by_letter
Out[111]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar',
'book']}
```

Se puede utilizar el método de diccionario `setdefault` para simplificar este flujo de trabajo. El bucle `for` anterior se podría reescribir así:

```
In [112]: by_letter = {}
```

```
In [113]:     for word in words:
.....:         letter = word[0]
.....:         by_letter.setdefault(letter, []).append(word)
.....:
```

```
In [114]: by_letter
Out[114]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar',
'book']}
```

El módulo `collections` integrado tiene una clase muy útil, `defaultdict`, que simplifica esto aún más. Para crearlo, se le pasa un tipo o una función para generar el valor predeterminado para cada espacio del diccionario:

```
In [115]: from collections import defaultdict
```

```
In [116]: by_letter = defaultdict(list)
```

```
In [117]:     for word in words:
.....:         by_letter[word[0]].append(word)
```

Tipos de claves de diccionario válidas

Aunque los valores de un diccionario pueden ser cualquier objeto Python, generalmente las claves tienen que ser objetos inmutables, como tipos escalares (`int`, `float`, `string`) o tuplas (todos los objetos de la tupla

tienen que ser también inmutables). Aquí el término técnico es *hashability*, o capacidad de resumen. Se puede verificar si un objeto es resumible o *hashable* (es decir, se puede usar como clave en un diccionario) con la función `hash`:

```
In [118]: hash("string")
Out[118]: 3634226001988967898
```

```
In [119]: hash((1, 2, (2, 3)))
Out[119]: -9209053662355515447
```

```
In [120]: hash((1, 2, [2, 3])) # falla porque las listas
son mutables
```

```
TypeError                                Traceback (most recent call
last)
<ipython-input-120-473c35a62c0b> in <module>
--> 1 hash((1, 2, [2, 3])) # falla porque las listas son
mutables
TypeError: unhashable type:
'list'
```

Los valores `hash` que se observan al emplear la función `hash` dependerán en general de la versión de Python con la que se esté trabajando.

Para utilizar una lista como una clave, una opción es convertirla en tupla, que se puede resumir tanto como sus elementos:

```
In [121]: d = {}

In [122]: d[tuple([1, 2, 3])] = 5

In [123]: d
Out[123]: {(1, 2, 3): 5}
```

Conjunto o *set*

Un conjunto o *set* es una colección desordenada de elementos únicos. Se pueden crear de dos maneras, mediante la función `set` o con un literal de conjunto con llaves:

```
In [124]: set([2, 2, 2, 1, 3, 3])
Out[124]: {1, 2, 3}
```

```
In [125]: {2, 2, 2, 1, 3, 3}
Out[125]: {1, 2, 3}
```

Los conjuntos soportan operaciones matemáticas de conjunto como unión, intersección, diferencia y diferencia simétrica. Veamos estos dos conjuntos de ejemplo:

```
In [126]: a = {1, 2, 3, 4, 5}
```

```
In [127]: b = {3, 4, 5, 6, 7, 8}
```

La unión de estos dos conjuntos es el conjunto de los distintos elementos que aparecen en cualquiera de los dos. Esto se puede codificar con el método `union` o con el operador binario `|`:

```
In [128]: a.union(b)
Out[128]: {1, 2, 3, 4, 5, 6, 7, 8}
```

```
In [129]: a | b
Out[129]: {1, 2, 3, 4, 5, 6, 7, 8}
```

La intersección contiene los elementos que aparecen en ambos conjuntos. Se pueden utilizar tanto el operador `&` como el método `intersection`:

```
In [130]: a.intersection(b)
Out[130]: {3, 4, 5}
```

```
In [131]: a & b
Out[131]: {3, 4, 5}
```

Véase en la tabla 3.1 una lista de los métodos de conjunto más utilizados.

Tabla 3.1. Operaciones con conjuntos de Python.

Función	Sintaxis alternativa	Descripción
---------	----------------------	-------------

<code>a.add(x)</code>	N/A	Suma el elemento x al conjunto a
<code>a.clear()</code>	N/A	Reinicia el conjunto a a un estado vacío, descartando todos sus elementos
<code>a.remove()</code>	N/A	Elimina el elemento x del conjunto a
<code>a.pop()</code>	N/A	Elimina un elemento arbitrario del conjunto a, produciendo un <code>KeyError</code> si el conjunto está vacío
<code>a.union(b)</code>	$a \mid b$	Todos los elementos únicos de a y b
<code>a.update(b)</code>	$a \mid= b$	Fija el contenido de a para que sea la unión de los elementos de a y b
<code>a.intersection(b)</code>	$a \& b$	Todos los elementos tanto de a como de b
<code>a.intersection_update(b)</code>	$a \&= b$	Fija el contenido de a para que sea la intersección de los elementos de a y b
<code>a.difference(b)</code>	$a - b$	Los elementos de a que no están en b
<code>a.difference_update(b)</code>	$a -= b$	Fija en a los elementos de a que no están en b
<code>a.symmetric_difference(b)</code>	$a \wedge b$	Todos los elementos de a o b pero no de los dos
<code>a.symmetric_difference_update(b)</code>	$a \wedge= b$	Fija a para que contenga los elementos de a o b pero no de ambos
<code>a.issubset(b)</code>	\leq	True si los elementos de a están todos contenidos en b
<code>a.issuperset(b)</code>	\geq	True si los elementos de b están todos contenidos en a
<code>a.isdisjoint(b)</code>	N/A	True si a y b no tienen elementos en común



Si se pasa una entrada que no es un conjunto a métodos como `union` e `intersection`, Python convertirá dicha entrada en un conjunto antes de ejecutar la operación. Al utilizar los operadores binarios, ambos objetos deben ser ya conjuntos.

Todas las operaciones lógicas de conjuntos disponen de equivalentes, que permiten reemplazar el contenido del conjunto en el lado izquierdo de

la operación por el resultado. Para conjuntos muy grandes, esto puede ser más eficiente:

```
In [132]: c = a.copy()
In [133]: c |= b
In [134]: c
Out[134]: {1, 2, 3, 4, 5, 6, 7, 8}
In [135]: d = a.copy()
In [136]: d &= b
In [137]: d
Out[137]: {3, 4, 5}
```

Al igual que las claves de diccionario, los elementos de conjunto deben ser en general inmutables, y deben ser además resumibles o *hashables* (lo que significa que llamar a hash en un valor no produce una excepción). Para almacenar elementos de estilo lista (u otras secuencias mutables) en un conjunto, se pueden convertir en tuplas:

```
In [138]: my_data = [1, 2, 3, 4]
In [139]: my_set = {tuple(my_data)}
In [140]: my_set
Out[140]: {(1, 2, 3, 4)}
```

También se puede comprobar si un conjunto es un subconjunto de otro conjunto (está contenido en él) o es un superconjunto de otro conjunto (es decir, contiene todos sus elementos):

```
In [141]: a_set = {1, 2, 3, 4, 5}
In [142]: {1, 2, 3}.issubset(a_set)
Out[142]: True
In [143]: a_set.issuperset({1, 2, 3})
Out[143]: True
```

Los conjuntos son iguales si y solo si sus contenidos son iguales:

```
In [144]: {1, 2, 3} == {3, 2, 1}
Out[144]: True
```

Funciones de secuencia integradas

Python tiene un montón de funciones de secuencia útiles con las que hay que familiarizarse y que hay que utilizar en cualquier oportunidad.

enumerate

Cuando se itera una secuencia, es habitual querer saber cuál es el índice del elemento actual. Un enfoque de aficionado sería algo así:

```
index = 0
for value in collection:
    # hace algo con value
    index += 1
```

Como esto es tan común, Python tiene una función integrada, `enumerate`, que devuelve una secuencia de tuplas (`i, value`):

```
for index, value in enumerate(collection):
    # hace algo con value
```

sorted

La función `sorted` devuelve una nueva lista ordenada a partir de los elementos de cualquier secuencia:

```
In [145]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[145]: [0, 1, 2, 2, 3, 6, 7]

In [146]: sorted("horse race")
Out[146]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

La función `sorted` acepta los mismos argumentos que el método `sort` en listas.

zip

La función `zip` «empareja» los elementos de una serie de listas, tuplas u otras secuencias para crear una lista de tuplas:

```
In [147]: seq1 = ["foo", "bar", "baz"]
In [148]: seq2 = ["one", "two", "three"]
In [149]: zipped = zip(seq1, seq2)
In [150]: list(zipped)
Out[150]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

`zip` puede tomar un número de secuencias aleatorias y el número de elementos que produce viene determinado por la secuencia más corta:

```
In [151]: seq3 = [False, True]
In [152]: list(zip(seq1, seq2, seq3))
Out[152]: [('foo', 'one', False), ('bar', 'two', True)]
```

Habitualmente se utiliza `zip` para iterar simultáneamente por varias secuencias, quizá también combinado con `enumerate`:

```
In [153]: for index, (a, b) in enumerate(zip(seq1, seq2)):
.....:     print(f"{index}: {a}, {b}")
.....:
0: foo, one
1: bar, two
2:     baz,
three
```

reversed

`reversed` itera por los elementos de una secuencia en orden inverso:

```
In [154]: list(reversed(range(10)))
Out[154]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```


Conviene recordar que `reversed` es un generador (concepto que veremos con más detalle más tarde), de modo que no crea la secuencia inversa hasta que se materializa (por ejemplo, con `list` o un bucle `for`).

Comprensiones de lista, conjunto y diccionario

Las comprensiones de lista son una característica del lenguaje Python cómoda y muy utilizada. Permiten formar de manera concisa una nueva lista filtrando los elementos de una colección, y transformando los elementos que pasan el filtro en una única expresión concisa. Toman la forma básica:

```
[expr for value in collection if condition]
```

Que es equivalente al siguiente bucle `for`:

```
result = []
for value in collection:
    if condition:
        result.append(expr)
```

La condición de filtro se puede omitir, dejando solo la expresión. Por ejemplo, dada una lista de cadenas de texto, podemos filtrar cadenas con longitud 2 o menor y convertirlas a mayúsculas de esta forma:

```
In [155]: strings = ["a", "as", "bat", "car", "dove",
                    "python"]
```

```
In [156]: [x.upper() for x in strings if len(x) > 2]
Out[156]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Las comprensiones de conjunto y diccionario son una extensión natural, produciendo de una forma idiomáticamente similar conjuntos y diccionarios en lugar de listas.

Una comprensión de diccionario es algo parecido a esto:

```
dict_comp = {key-expr: value-expr for value in collection
             if condition}
```

Una comprensión de conjunto es similar a la comprensión de lista equivalente, salvo que lleva llaves en lugar de paréntesis cuadrados:

```
set_comp = {expr for value in collection if condition}
```

Al igual que las comprensiones de lista, las de conjunto y diccionario pueden facilitar de forma similar la escritura y lectura.

Veamos la lista de cadenas de texto de antes. Supongamos que queremos un conjunto que contenga solamente las longitudes de las cadenas contenidas en la colección; podríamos codificar esto fácilmente con una comprensión de conjunto:

```
In [157]: unique_lengths = {len(x) for x in strings}
```

```
In [158]: unique_lengths  
Out[158]: {1, 2, 3, 4, 6}
```

También podríamos expresar esto de una manera más funcional con la función `map`, que presentaremos en breve:

```
In [159]: set(map(len, strings))  
Out[159]: {1, 2, 3, 4, 6}
```

Como un ejemplo sencillo de comprensión de diccionario, podríamos crear un mapa de consulta de estas cadenas de texto para hallar sus ubicaciones en la lista:

```
In [160]: loc_mapping = {value: index for index, value in  
enumerate(strings)}  
  
In [161]: loc_mapping  
Out[161]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4,  
'python': 5}
```

Comprensiones de lista anidadas

Supongamos que tenemos una lista de listas que contiene algunos nombres en inglés y español:

```
In [162]: all_data = ["John", "Emily", "Michael", "Mary",
```

```
"Steven"],  
.....: ["Maria", "Juan", "Javier", "Natalia", "Pilar"]]
```

Imaginemos que queremos conseguir una sola lista que contenga todos los nombres con dos o más a. Podríamos sin duda hacer esto con un sencillo bucle for:

```
In [163]: names_of_interest = []
```

```
In [164]: for names in all_data:  
.....:     enough_as = [name for name in names if  
.....:                  name.count("a") >= 2]  
.....:     names_of_interest.extend(enough_as)  
.....:
```

```
In [165]: names_of_interest  
Out[165]: ['Maria', 'Natalia']
```

Se podría realizar esta operación por completo con una única comprensión de lista anidada, que tendría un aspecto similar a este:

```
In [166]: result = [name for names in all_data for name in  
names  
.....:                if name.count("a") >= 2]
```

```
In [167]: result  
Out[167]: ['Maria', 'Natalia']
```

Al principio, las comprensiones de lista anidadas son un poco difíciles de entender. Las partes for de la comprensión de lista se organizan de acuerdo con el orden de anidamiento, y cualquier condición de filtro se coloca al final como antes. Aquí tenemos otro ejemplo, en el que «reducimos» una lista de tuplas de enteros a una sencilla lista de enteros:

```
In [168]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

```
In [169]: flattened = [x for tup in some_tuples for x in  
tup]
```

```
In [170]: flattened
Out[170]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Hay que recordar que el orden de las expresiones `for` sería el mismo si se escribiera un bucle `for` anidado en lugar de una comprensión de lista:

```
flattened = []
for tup in some_tuples:
    for x in tup:
        flattened.append(x)
```

Se pueden tener de forma arbitraria muchos niveles de anidamiento, aunque si se tienen más de dos o tres, es probable que deje de tener sentido desde el punto de vista de la legibilidad del código. Es importante distinguir la sintaxis que acabamos de mostrar de la de una comprensión de lista dentro de una comprensión de lista, que también es perfectamente válida:

```
In [172]: [[x for x in tup] for tup in some_tuples]
Out[172]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Esto produce una lista de listas, en lugar de una lista reducida de todos los elementos internos.

3.2 Funciones

Las funciones son el método principal y más importante de Python para organizar y reutilizar código. Como regla general, si nos anticipamos a la necesidad de repetir el mismo código o uno muy parecido más de una vez, puede merecer la pena escribir una función que se pueda reutilizar. Las funciones también pueden ayudar a que el código sea más legible dando un nombre a un grupo de sentencias Python.

Las funciones se declaran con la palabra clave `def`. Una función contiene un bloque de código, con un uso opcional de la palabra clave `return`:

```
In [173]: def my_function(x, y):  
.....:         return x + y
```

Cuando se alcanza una línea con `return`, el valor o la expresión que venga después se envía al contexto en el que se llamó a la función, por ejemplo:

```
In [174]: my_function(1, 2)  
Out[174]: 3
```

```
In [175]: result = my_function(1, 2)
```

```
In [176]: result  
Out[176]: 3
```

No hay problema por tener varias sentencias `return`. Si Python alcanza el final de una función sin encontrar una sentencia `return`, devuelve automáticamente `None`. Por ejemplo:

```
In [177]: def function_without_return(x):  
.....:         print(x)
```

```
In [178]: result = function_without_return("hello!")  
hello!
```

```
In [179]: print(result)
```

```
None
```

Cada función puede tener argumentos posicionales y de palabra clave. Los argumentos de palabra clave son los más utilizados para especificar valores predeterminados u argumentos opcionales.

Aquí definiremos una función con un argumento opcional `z` que tiene el valor predeterminado de `1.5`:

```
def my_function2(x, y, z=1.5):  
    if z > 1:  
        return z * (x + y)  
    else:
```

```
return z / (x + y)
```

Los argumentos de palabra clave son opcionales, pero se deben especificar todos los argumentos posicionales al llamar a una función.

Se pueden pasar valores al argumento `z` incluyendo o no la palabra clave, aunque es recomendable utilizarla:

```
In [181]: my_function2(5, 6, z=0.7)
```

```
Out[181]: 0.06363636363636363
```

```
In [182]: my_function2(3.14, 7, 3.5)
```

```
Out[182]: 35.49
```

```
In [183]: my_function2(10, 20)
```

```
Out[183]: 45.0
```

La principal restricción en los argumentos de función es que los argumentos de palabra clave deben seguir a los posicionales (si los hay). Los argumentos de palabra clave se pueden especificar en cualquier orden, lo cual nos libera de tener que recordar el orden en el que se especificaron. Basta con acordarse de cuáles son sus nombres.

Espacios de nombres, ámbito y funciones locales

Las funciones pueden acceder a variables creadas dentro de la misma función, así como a las que están fuera de la función en ámbitos más elevados (o incluso globales). En Python, un espacio de nombres es otra forma de denominar a un ámbito de variable, además de que lo describe mejor.

Cualquier variable asignada dentro de una función de forma predeterminada está asignada asimismo al espacio de nombres local. Cuando se llama a la función, se crea el espacio de nombres, que se llena inmediatamente con los argumentos de la función. Una vez finalizada esta, el espacio de nombres local se destruye (con algunas excepciones que quedan fuera del alcance de este libro). Veamos la siguiente función:

```
def func():
```

```
a = []  
for i in range(5):  
    a.append(i)
```

Cuando se llama a la función `func()`, se crea la lista vacía `a`, se añaden cinco elementos y después `a` es destruida cuando la función sale. Supongamos que lo que habíamos hecho era declarar `a` de la siguiente forma:

```
In [184]: a = []
```

```
In [185]: def func():  
.....:         for i in range(5):  
.....:         a.append(i)
```

Cada llamada a `func` modificará la lista `a`:

```
In [186]: func()
```

```
In [187]: a  
Out[187]: [0, 1, 2, 3, 4]
```

```
In [188]: func()
```

```
In [189]: a  
Out[189]: [0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
```

Es posible asignar variables fuera del ámbito de la función, pero dichas variables deben estar declaradas explícitamente utilizando las palabras clave `global` o `nonlocal`:

```
In [190]: a = None
```

```
In [191]: def bind_a_variable():  
.....:         global a  
.....:         a = []  
.....:         bind_a_variable()  
.....:
```

```
In [192]: print(a)
```

```
[]
```

`nonlocal` permite a una función modificar variables definidas en un ámbito de mayor nivel que no es global. Como su uso es algo esotérico (nunca lo utilizo en este libro), para más información conviene consultar la documentación de Python.



No es aconsejable usar la palabra clave `keyword`. Normalmente, se utilizan variables globales para almacenar cierto tipo de estado en un sistema. Si uno se da cuenta de que utiliza muchas, quizá ello esté indicando la necesidad de programación orientada a objetos (utilizando clases).

Devolver varios valores

Cuando empecé a programar por primera vez en Python, después de haber programado en Java y C++, una de mis características favoritas era la capacidad para devolver varios valores desde una función con una sintaxis muy sencilla. Aquí tenemos un ejemplo:

```
def f():  
    a = 5  
    b = 6  
    c = 7  
    return a, b, c  
a, b, c = f()
```

En análisis de datos y otras aplicaciones científicas se hace esto a menudo. Lo que está ocurriendo aquí es que la función devuelve realmente solo un objeto, una tupla, que después se desempaqueta para obtener las variables del resultado. En el ejemplo anterior podríamos haber hecho esto:

```
return_value = f()
```

En este caso, `return_value` sería una tupla de tres, conteniendo las tres variables devueltas. Una alternativa a devolver varios valores del modo que hemos visto, que quizá resulte interesante, sería devolver un diccionario:


```
def f():
    a = 5
    b = 6
    c = 7
    return {"a" : a, "b" : b, "c" : c}
```

Esta técnica alternativa puede resultar útil, dependiendo de lo que se esté tratando de hacer.

Las funciones son objetos

Como las funciones Python son objetos, muchas construcciones que son difíciles de codificar en otros lenguajes se pueden expresar aquí fácilmente. Supongamos que estamos haciendo limpieza de datos y necesitamos aplicar unas cuantas transformaciones a la siguiente lista de cadenas de texto:

```
In [193]: states = [" Alabama ", "Georgia!", "Georgia",
.....:             "georgia", "FlOrIda",
.....:             " south carolina###", "West virginia?"]
```

Cualquiera que haya trabajado alguna vez con datos de encuestas enviados por usuarios habrá visto unos resultados desordenados como estos. Muchas cosas tienen que ocurrir para que esta lista de cadenas de texto sea uniforme y esté lista para ser analizada: eliminar espacios en blanco, quitar signos de puntuación y estandarizar mayúsculas y minúsculas. Una forma de hacerlo es utilizando métodos de cadena de texto internos junto con el módulo de librería estándar `re` para las expresiones regulares:

```
import re
def clean_strings(strings):
    result = []
    for value in strings:
        value = value.strip()
        value = re.sub("[!#?]", "", value)
        value = value.title()
        result.append(value)
    return result
```

El resultado tiene este aspecto:

```
In [195]: clean_strings(states)
Out[195]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

Un método alternativo que puede resultar útil es crear una lista de las operaciones que se desean aplicar a un determinado conjunto de cadenas de texto:

```
def remove_punctuation(value):
    return re.sub("[!#?]", "", value)
clean_ops = [str.strip, remove_punctuation, str.title]
def clean_strings(strings, ops):
    result = []
    for value in strings:
        for func in ops:
            value = func(value)
        result.append(value)
    return result
```

Entonces tenemos lo siguiente:

```
In [197]: clean_strings(states, clean_ops)
Out[197]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

Un patrón más funcional como este permite modificar fácilmente el modo en que las cadenas de texto se transforman a un nivel muy alto. La función `clean_strings` es también ahora más reutilizable y genérica.

Se pueden utilizar funciones como argumentos para otras funciones, como la función integrada `map`, que aplica una función a una secuencia de algún tipo:

```
In [198]: for x in map(remove_punctuation, states):
.....:         print(x)
Alabama
Georgia
Georgia
georgia
FlOrIda
south carolina
West virginia
```

`map` se puede usar como alternativa a las comprensiones de lista sin ningún tipo de filtro.

Funciones anónimas (lambda)

Python soporta las denominadas funciones anónimas o `lambda`, una forma de escribir funciones que consisten en una única sentencia, el resultado de la cual es el valor devuelto. Se definen con la palabra clave `lambda`, cuyo único significado es «estamos declarando una función anónima»:

```
In [199]: def short_function(x):
.....:         return x * 2

In [200]: equiv_anon = lambda x: x * 2
```

Normalmente las denominaré funciones `lambda` en el resto del libro. Son especialmente cómodas en análisis de datos porque, como veremos, hay muchos casos en los que las funciones de transformación de datos tomarán

funciones como argumentos. Con frecuencia suele ser más rápido (y claro) pasar una función lambda, a diferencia de escribir una declaración de función completa o incluso asignar la función lambda a una variable local. Veamos este ejemplo:

```
In [201]:      def apply_to_list(some_list, f):
.....:         return [f(x) for x in some_list]

In [202]:      ints = [4, 0, 1, 5, 6]

In [203]:      apply_to_list(ints, lambda x: x * 2)
Out[203]:      [8, 0, 2, 10, 12]
```

También podríamos haber escrito `[x * 2 for x in ints]`, pero aquí tendríamos la posibilidad de pasarle un operador personalizado a la función `apply_to_list`.

Como un ejemplo más, supongamos que queremos ordenar una colección de cadenas de texto por el número de las letras de que se compone cada cadena:

```
In [204]: strings = ["foo", "card", "bar", "aaaa", "abab"]
```

Podríamos pasar una función lambda al método `sort` de la lista:

```
In [205]: strings.sort(key=lambda x: len(set(x)))

In [206]: strings
Out[206]: ['aaaa', 'foo', 'abab', 'bar', 'card']
```

Generadores

Muchos objetos en Python soportan iteración, como, por ejemplo, sobre los objetos de una lista o sobre las líneas de un archivo. Esto se lleva a cabo por medio del protocolo iterador, una forma genérica de hacer que los objetos sean iterables. Por ejemplo, iterar sobre un diccionario produce las claves de diccionario:

```
In [207]: some_dict = {"a": 1, "b": 2, "c": 3}
```

```
In [208]: for key in some_dict:
.....:     print(key)
a
b
c
```

Al escribir `for key in some_dict`, el intérprete de Python intenta primero crear un iterador de `some_dict`:

```
In [209]: dict_iterator = iter(some_dict)
```

```
In [210]: dict_iterator
```

```
Out[210]: <dict_keyiterator at 0x7fefe45465c0>
```

Un iterador es cualquier objeto que le proporcionará otros objetos al intérprete de Python utilizado en un contexto como, por ejemplo, un bucle `for`. La mayor parte de los métodos que esperan una lista (o un objeto similar a una lista) aceptarán también cualquier objeto iterable, incluyendo métodos integrados como `min`, `max` y `sum`, y constructores de tipo como `list` y `tuple`:

```
In [211]: list(dict_iterator)
Out[211]: ['a', 'b', 'c']
```

Un generador es una forma cómoda, parecida a escribir una función normal, de construir un nuevo objeto iterable. Mientras las funciones normales ejecutan y devuelven un solo resultado a la vez, los generadores pueden devolver una secuencia de varios valores parando y siguiendo con la ejecución cada vez que se utiliza el generador. Para crear uno, es mejor usar la palabra clave `yield` en lugar de `return` en una función:

```
def squares(n=10):
    print(f"Generating squares from 1 to {n ** 2}")
    for i in range(1, n + 1):
        yield i ** 2
```

En realidad, cuando se llama al generador, no se ejecuta inmediatamente ningún código:

```
In [213]: gen = squares()
```

```
In [214]: gen
```

```
Out[214]: <generator object squares at 0x7fefe437d620>
```

No es hasta que se le piden elementos al generador cuando empieza a ejecutar su código:

```
In [215]:          for x in gen:
.....:          print(x, end=" ")
Generating squares from 1 to 100
1 4 9 16 25 36 49 64 81 100
```



Como los generadores producen resultados de un elemento cada vez frente a una lista entera de una sola vez, los programas en los que se emplean utilizan menos memoria.

Expresiones generadoras

Otra forma de crear un generador es utilizando una expresión generadora, que es un generador análogo a las comprensiones de lista, diccionario y conjunto. Para crear uno, encerramos lo que de otro modo sería una comprensión de lista dentro de paréntesis en lugar de llaves:

```
In [216]: gen = (x ** 2 for x in range(100))
```

```
In [217]: gen
```

```
Out[217]: <generator object <genexpr> at 0x7fefe437d000>
```

Esto es equivalente al siguiente generador, que incluye más palabras:

```
def _make_gen():
    for x in range(100):
        yield x ** 2
gen = _make_gen()
```

Las expresiones generadoras se pueden emplear en lugar de las comprensiones de lista como argumentos de función en algunos casos:

```
In [218]: sum(x ** 2 for x in range(100))
Out[218]: 328350
```

```
In [219]: dict((i, i ** 2) for i in range(5))
Out[219]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Dependiendo del número de elementos producidos por la expresión de la comprensión, algunas veces la versión generadora puede ser notablemente más rápida.

Módulo itertools

El módulo `itertools` de la librería estándar tiene una colección de generadores para muchos algoritmos de datos habituales. Por ejemplo, `groupby` toma cualquier secuencia y una función, agrupando los elementos consecutivos de la secuencia por el valor que devuelve la función.

Aquí tenemos un ejemplo:

```
In [220]: import itertools
```

```
In [221]: def first_letter(x):
.....:     return x[0]
```

```
In [222]: names = ["Alan", "Adam", "Wes", "Will", "Albert",
"Steven"]
```

```
In [223]: for letter, names in itertools.groupby(names,
first_letter):
.....:     print(letter, list(names)) # names es un
generador
A    ['Alan', 'Adam']
W    ['Wes', 'Will']
A    ['Albert']
S    ['Steven']
```

Consulte en la tabla 3.2 una lista de algunas funciones más de `itertools` que me han resultado muchas veces útiles.

Tabla 3.2. Algunas funciones útiles de `itertools`.

Función	Descripción
<code>chain(*iterables)</code>	Genera una secuencia encadenando iteradores. Una vez se han agotado los elementos del primer iterador, se devuelven los elementos del siguiente, y así sucesivamente
<code>combinations(iterable, k)</code>	Genera una secuencia de todas las posibles tuplas de <code>k</code> elementos en el iterable, ignorando el orden y sin reemplazar (vea también la función acompañante <code>combinations_with_replacement</code>)
<code>permutations(iterable, k)</code>	Genera una secuencia de todas las posibles tuplas de <code>k</code> elementos en el iterable, respetando el orden
<code>groupby(iterable[, keyfunc])</code>	Genera <code>(key, sub-iterator)</code> para cada clave única
<code>product(*iterables, repeat=1)</code>	Genera el producto cartesiano de los iterables de entrada como tuplas, similar a un bucle <code>for</code> anidado



Puede que le convenga revisar en la documentación oficial de Python (<https://docs.python.org/3/library/itertools.html>) más información sobre este práctico módulo integrado.

Errores y manejo de excepciones

Manejar los errores o excepciones de Python con elegancia es una parte importante de la creación de programas robustos. En aplicaciones de análisis de datos, muchas funciones solo admiten ciertos tipos de entrada. Por ejemplo, la función `float` de Python puede convertir una cadena de texto en un número de punto flotante, pero produce un `ValueError` con entradas no adecuadas:

```
In [224]: float("1.2345")
Out[224]: 1.2345
```



```
In [225]: float("something")
```

```
ValueError      Traceback (most recent call last)
<ipython-input-225-5ccfe07933f4> in <module>
--> 1 float("something")
ValueError: could not convert string to float: 'something'
```

Supongamos que queremos una versión de float que falle dignamente, devolviendo el argumento de entrada. Podemos hacerlo escribiendo una función que encierre la llamada a float en un bloque try/except (ejecuta este código en IPython):

```
def attempt_float(x):
    try:
        return float(x)
    except:
        return x
```

El código de la parte except del bloque solo se ejecutará si float(x) produce una excepción:

```
In [227]: attempt_float("1.2345")
Out[227]: 1.2345

In [228]: attempt_float("something")
Out[228]: 'something'
```

Quizá se haya dado cuenta de que float puede producir excepciones distintas a ValueError:

```
In [229]: float((1, 2))
```

```
TypeError      Traceback (most recent call last)
<ipython-input-229-82f777b0e564> in <module>
--> 1 float((1, 2))
TypeError: float() argument must be a string or a real
number, not 'tuple'
```

También es posible que resulte interesante suprimir solamente `ValueError`, pues un `TypeError` (la entrada no era una cadena de texto o un valor numérico) podría indicar un error legítimo en el programa. Para ello, escribimos el tipo de excepción después de `except`:

```
def attempt_float(x):  
    try:  
        return float(x)  
    except ValueError:  
        return x
```

Entonces tenemos:

```
In [231]: attempt_float((1, 2))
```

```
TypeError          Traceback (most recent call last)  
<ipython-input-231-8b0026e9e6b7> in <module>  
--> 1 attempt_float((1, 2))  
<ipython-input-230-6209ddec2b5> in attempt_float(x)  
      1     def attempt_float(x):  
      2     try:  
-->      3     return float(x)  
      4     except ValueError:  
      5     return x  
TypeError: float() argument must be a string or a real  
number, not 'tuple'
```

Se pueden capturar varios tipos de excepción escribiendo una tupla de tipos de excepción en su lugar (los paréntesis son necesarios):

```
def attempt_float(x):  
    try:  
        return float(x)  
    except (TypeError, ValueError):  
        return x
```

En algunos casos, quizá no interese suprimir una excepción, pero sí que se ejecute cierto código sin tener en cuenta si el código del bloque `try` funciona o no. Para ello utilizamos finalmente:

```
f = open(path, mode="w")
try:
    write_to_file(f)
finally:
    f.close()
```

En este caso el objeto archivo `f` siempre se cerrará. De forma similar, se puede tener código que se ejecute solamente si el bloque `try:` funciona utilizando `else:`

```
f = open(path, mode="w")
try:
    write_to_file(f)
except:
    print("Failed")
else:
    print("Succeeded")
finally:
    f.close()
```

Excepciones en IPython

Si se produce una excepción mientras se ejecuta un *script* con `%run` o cualquier sentencia, IPython imprimirá de forma predeterminada un *traceback* (o seguimiento de pila de llamadas) completo con algunas líneas de contexto alrededor de la posición en cada punto de la pila:

```
In [10]: %run examples/ipython_bug.py
-----
AssertionError      Traceback (most recent call last)
/home/wesm/code/pydata-book/examples/ipython_bug.py      in
<module>()
      13          throws_an_exception()
```

```

14
--> 15      calling_things()
/home/wesm/code/pydata-book/examples/ipython_bug.py      in
calling_things()
11      def calling_things():
12      works_fine()
--> 13      throws_an_exception()
14
15      calling_things()
/home/wesm/code/pydata-book/examples/ipython_bug.py      in
throws_an_exception()
7      a = 5
8      b = 6
—> 9      assert(a + b == 10)
10
11      def calling_things():
AssertionError:

```

Tener contexto adicional es ya de por sí una gran ventaja con respecto al intérprete de Python estándar (que no lo ofrece). Se puede controlar la cantidad de contexto mostrado con el comando mágico `%xmode`, desde `Plain` (igual que en el intérprete de Python estándar) hasta `Verbose` (que añade contexto incluso entre los valores de argumento de función y mucho más). Como veremos después en el apéndice B, se puede acceder a la pila (usando los comandos mágicos `%debug` o `%pdb`) después de producirse un error para realizar depuración interactiva a posteriori.

3.3 Archivos y el sistema operativo

La mayor parte de este libro utiliza herramientas de alto nivel, como `pandas.read_csv`, para leer archivos de datos del disco y convertirlos en estructuras de datos de Python. Sin embargo, es importante comprender los fundamentos del trabajo con archivos en Python. Por suerte, es relativamente sencillo de entender, una de las razones por las que Python es tan popular para procesamiento de texto y archivos.

Para abrir un archivo para su lectura o escritura, utilizamos la función integrada `open` con una ruta de archivos relativa o absoluta y una codificación de archivos opcional:

```
In [233]: path = "examples/segismundo.txt"
```

```
In [234]: f = open(path, encoding="utf-8")
```

Aquí la costumbre es pasar `encoding="utf-8"`, porque la codificación Unicode predeterminada para leer archivos varía de una plataforma a otra.

Por omisión, el archivo se abre en el modo de solo lectura `"r"`. Podemos después tratar el objeto de archivo `f` como una lista e iterar sobre las líneas de este modo:

```
for line in f:
    print(line)
```

Las líneas salen del archivo con los marcadores de final de línea (EOL: end-of-line) intactos, de modo que normalmente veremos código para obtener una lista de líneas libre de EOL en un archivo como el siguiente:

```
In [235]: lines = [x.rstrip() for x in open(path,
encoding="utf-8")]
```

```
In [236]: lines
```

```
Out[236]:
```

```
['Sueña el rico en su riqueza,',
'que más cuidados le ofrece;',
'',
'sueña el pobre que padece',
'su miseria y su pobreza;',
'',
'sueña el que a medrar empieza,',
'sueña el que afana y pretende,',
'sueña el que agravia y ofende,',
'',
'y en el mundo, en conclusión,',
'todos sueñan lo que son,',
'aunque ninguno lo entiende.',
```

```
'']
```

Cuando se utiliza `open` para crear objetos de archivo, es recomendable cerrar el archivo cuando se haya terminado con él. Así se liberan sus recursos de nuevo para el sistema operativo:

```
In [237]: f.close()
```

Una de las formas de facilitar la limpieza de archivos abiertos es emplear la sentencia `with`:

```
In [238]: with open(path, encoding="utf-8") as f:
.....:     lines = [x.rstrip() for x in f]
```

Así se cerrará automáticamente el archivo `f` al salir del bloque `with`. No lograr asegurar que los archivos están cerrados no causará problemas en muchos programas o *scripts* pequeños, pero puede ser un problema en programas que necesiten interactuar con un gran número de archivos.

Si hubiéramos escrito `f = open(path, "w")`, se habría creado un nuevo archivo en `examples/segismundo.txt` (¡hay que tener cuidado!), sobrescribiendo un posible archivo ya existente. También está el modo de archivo “x”, que crea un archivo con permiso de escritura, pero da error si la ruta del archivo ya existe. Véase en la tabla 3.3 una lista de los modos válidos de lectura/escritura de archivos.

Tabla 3.3. Modos de archivo de Python.

Modo	Descripción
r	Modo de solo lectura
w	Modo de solo escritura; crea un nuevo archivo (borrando los datos de cualquier archivo con el mismo nombre)
x	Modo de solo escritura; crea un nuevo archivo pero da error si la ruta del archivo ya existe
a	Añade al archivo existente (crea el archivo si no existe)

r+	Lectura y escritura
b	Se suma al modo para trabajar con archivos binarios (por ejemplo, "rb" o "wb")
t	Modo de texto para archivos (decodificando automáticamente los bytes a Unicode); es el modo predeterminado si no se especifica

Para archivos con permiso de lectura, algunos de los métodos más usados son `read`, `seek` y `tell`. `read` devuelve un cierto número de caracteres del archivo. Lo que constituye un «carácter» viene determinado por la codificación del archivo o simplemente por los bytes sin procesar si el archivo se abre en modo binario:

```
In [239]: f1 = open(path)

In [240]: f1.read(10)
Out[240]: 'Sueña el r'

In [241]: f2 = open(path, mode="rb") # Modo binario

In [242]: f2.read(10)
Out[242]: b'Sue\xc3\xb1a el '
```

El método `read` avanza la posición del objeto de archivo por el número de bytes leídos. `tell` proporciona la posición actual:

```
In [243]: f1.tell()
Out[243]: 11

In [244]: f2.tell()
Out[244]: 10
```

Aunque leamos 10 caracteres del archivo `f1` abierto en modo texto, la posición es 11 porque hicieron falta todos esos bytes para decodificar 10 caracteres utilizando la codificación predeterminada. Se puede comprobar la codificación predeterminada en el módulo `sys`:

```
In [245]: import sys

In [246]: sys.getdefaultencoding()
Out[246]: 'utf-8'
```

Para obtener un comportamiento consistente a lo largo de las distintas plataformas, es mejor pasar una codificación (como `encoding="utf-8"`, ampliamente usada) al abrir archivos.

`seek` cambia la posición del archivo al byte indicado en el mismo:

```
In [247]: f1.seek(3)
Out[247]: 3
```

```
In [248]: f1.read(1)
Out[248]: 'ñ'
```

```
In [249]: f1.tell()
Out[249]: 5
```

Por último, nos acordamos de cerrar los archivos:

```
In [250]: f1.close()
```

```
In [251]: f2.close()
```

Para escribir texto en un archivo, se pueden usar los métodos `write` o `writelines` del archivo. Por ejemplo, podríamos crear una versión de `examples/segismundo.txt` sin líneas en blanco de la siguiente manera:

```
In [252]: path
Out[252]: 'examples/segismundo.txt'
```

```
In [253]: with open("tmp.txt", mode="w") as handle:
.....:     handle.writelines(x for x in open(path) if
.....:         len(x) > 1)
```

```
In [254]: with open("tmp.txt") as f:
.....:     lines = f.readlines()
```

```
In [255]: lines
Out[255]:
['Sueña el rico en su riqueza,\n',
'que más cuidados le ofrece;\n',
'sueña el pobre que padece\n',
'su miseria y su pobreza;\n',
```



```
'sueña el que a medrar empieza,\n',
'sueña el que afana y pretende,\n',
'sueña el que agravia y ofende,\n',
'y en el mundo, en conclusión,\n',
'todos sueñan lo que son,\n',
'aunque ninguno lo entiende.\n']
```

Consulte en la tabla 3.4 muchos de los métodos de archivo habitualmente utilizados.

Tabla 3.4. Métodos o atributos de archivo importantes de Python.

Método/atributo	Descripción
<code>read([size])</code>	Devuelve datos del archivo como bytes o como cadena de texto dependiendo del modo de archivo, indicando el argumento opcional <code>size</code> el número de bytes o caracteres de cadena de texto que hay que leer
<code>readable()</code>	Devuelve <code>True</code> si el archivo soporta operaciones <code>read</code>
<code>readlines([size])</code>	Devuelve una lista de líneas del archivo, con el argumento opcional <code>size</code>
<code>write(string)</code>	Escribe la cadena de texto pasada en el archivo
<code>writable()</code>	Devuelve <code>True</code> si el archivo soporta operaciones <code>write</code>
<code>writelines(strings)</code>	Escribe la secuencia de cadenas de texto pasada en el archivo
<code>close()</code>	Cierra el objeto de archivo
<code>flush()</code>	Vacía el buffer de entrada/salida interno en el disco
<code>seek(pos)</code>	Va a la posición indicada del archivo (entero)
<code>seekable()</code>	Devuelve <code>True</code> si el objeto de archivo soporta búsquedas y, por lo tanto, acceso aleatorio (algunos objetos de tipo archivo no lo soportan)
<code>tell()</code>	Devuelve la posición actual del archivo como entero
<code>closed</code>	<code>True</code> si el archivo está cerrado
<code>encoding</code>	La codificación empleada para interpretar los bytes del archivo como Unicode (normalmente UTF-8)

Bytes y Unicode con archivos

El comportamiento predeterminado para los archivos de Python (ya sean de lectura o escritura) es el modo de texto, lo cual significa que el objetivo es trabajar con cadenas de texto Python (por ejemplo, Unicode). Esto contrasta con el modo binario, que se puede conseguir añadiendo `b` al modo del archivo. Recuperando el archivo de la sección anterior (que contiene caracteres no ASCII con codificación UTF-8), tenemos:

```
In [258]: with open(path) as f:
.....:         chars = f.read(10)
```

```
In [259]: chars
Out[259]: 'Sueña el r'
```

```
In [260]: len(chars)
Out[260]: 10
```

UTF-8 es una codificación Unicode de longitud variable, de modo que al pedir un cierto número de caracteres del archivo, Python lee de dicho archivo los bytes suficientes (que podrían ser tan pocos como 10 o tantos como 40) para decodificar todos esos caracteres. Pero si se abre el archivo en el modo “rb”, read pide ese número exacto de bytes:

```
In [261]:         with open(path, mode="rb") as f:
.....:             data = f.read(10)
```

```
In [262]: data
Out[262]: b'Sue\xc3\xb1a el '
```

Dependiendo de la codificación del texto, se podrían decodificar los bytes a un objeto `str`, pero solo si cada uno de los caracteres Unicode codificados está totalmente formado:

```
In [263]: data.decode("utf-8")
Out[263]: 'Sueña el '
```

```
In [264]: data[:4].decode("utf-8")
```

```
UnicodeDecodeError      Traceback (most recent call last)
<ipython-input-264-846a5c2fed34> in <module>
--> 1 data[:4].decode("utf-8")
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in
position 3: unexpected
end of data
```

El modo de texto, combinado con la opción encoding de open, ofrece una forma conveniente de convertir de una codificación Unicode a otra:

```
In [265]: sink_path = "sink.txt"
```

```
In [266]: with open(path) as source:
.....:     with open(sink_path, "x", encoding="iso-8859-1")
.....:         as sink:
.....:         sink.write(source.read())
```

```
In [267]: with open(sink_path, encoding="iso-8859-1") as f:
.....:         print(f.read(10))
Sueña el r
```

Hay que tener cuidado al utilizar seek cuando se abren archivos en cualquier modo que no sea binario. Si la posición del archivo cae en medio de los bytes que definen un carácter Unicode, entonces las posteriores lecturas darán error:

```
In [269]: f = open(path, encoding='utf-8')
```

```
In [270]: f.read(5)
Out[270]: 'Sueña'
```

```
In [271]: f.seek(4)
Out[271]: 4
```

```
In [272]: f.read(1)
```

```
UnicodeDecodeError      Traceback (most recent call last)
<ipython-input-272-5a354f952aa4> in <module>
--> 1 f.read(1)
```

```

/miniconda/envs/book-env/lib/python3.10/codecs.py      in
decode(self, input, final)
    320         # decodifica la entrada (teniendo en
                cuenta el búfer)
    321         data = self.buffer + input
->         (result, consumed) =
    322         self._buffer_decode(data, self.errors,
                final
    )
    323         # mantiene sin decodificar la entrada
                hasta la siguiente llamada
    324         self.buffer = data[consumed:]
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb1 in
position 0: invalid start byte

```

```
In [273]: f.close()
```

Si le parece que va a realizar regularmente análisis de datos con datos de texto no ASCII, dominar la funcionalidad Unicode de Python le resultará de gran utilidad. Consulte la documentación en línea de Python (<https://docs.python.org>) para obtener más información.

3.4 Conclusión

Ahora que ya tenemos parte de los fundamentos del entorno y lenguaje Python bajo control, ha llegado la hora de avanzar y aprender NumPy y la computación orientada a arrays en Python.

Fundamentos de NumPy: arrays y computación vectorizada

NumPy, abreviatura de Numerical Python, es uno de los paquetes básicos más importantes para cálculo numérico en Python. Muchos paquetes computacionales que ofrecen funcionalidad científica emplean los objetos array de NumPy como una de las lenguas vehiculares estándares para intercambio de datos. Buena parte del conocimiento que vamos a tratar aquí sobre NumPy puede aplicarse igualmente a pandas.

Estas son algunas de las características que encontramos en NumPy:

- `ndarray`, un eficiente array multidimensional que ofrece rápidas operaciones aritméticas orientadas a arrays y unas capacidades de difusión muy flexibles.
- Funciones matemáticas para operaciones rápidas con arrays enteros de datos sin tener que escribir bucles.
- Herramientas para leer o escribir datos de array en disco y trabajar con archivos proyectados en memoria.
- Una API escrita en C para conectar NumPy con librerías escritas en C, C++ o FORTRAN.

Como NumPy proporciona una API en C completa y bien documentada, resulta muy sencillo pasar datos a librerías externas escritas en un lenguaje de bajo nivel, al igual que a dichas librerías les resulta fácil devolver datos a Python como arrays de NumPy. Esta característica ha convertido a Python en el lenguaje elegido para contener bases de código heredadas de C, C++ o FORTRAN y darles una interfaz dinámica y accesible.

Aunque NumPy como tal no ofrece funcionalidad de modelado o científica, entender bien los arrays de NumPy y la computación orientada a

los mismos permite utilizar herramientas con semántica de cálculo de arrays, como pandas, de una manera mucho más eficaz. Como NumPy es un tema de gran envergadura, trataremos con más detalle muchas funciones avanzadas de NumPy, como la difusión, en el apéndice A. Muchas de ellas no son necesarias para seguir el resto del libro, pero pueden ayudar a profundizar más en la ciencia computacional con Python.

Para la mayoría de las aplicaciones de análisis de datos, las principales áreas de funcionalidad en las que nos centraremos son las siguientes:

- Operaciones rápidas basadas en arrays para realizar cálculos con datos, como procesado y limpieza, hacer subconjuntos y filtrado, transformaciones y cualquier otro tipo de cálculo.
- Algoritmos de arrays habituales, como ordenación, unique y operaciones de conjuntos.
- Eficaces estadísticas descriptivas y agregación o resumen de datos.
- Alineación de datos y manipulaciones de datos relacionales para combinar conjuntos de datos heterogéneos.
- Expresar lógica condicional como expresiones de array en lugar de usar bucles con estructuras if-elif-else.
- Manipulaciones de datos válidas para grupos (agregación, transformación y aplicación de funciones).

Aunque NumPy ofrece un buen fundamento para procesar datos numéricos de forma genérica, muchos lectores preferirán usar pandas como base para la mayoría de los tipos de estadísticas o análisis, especialmente con datos tabulares. Asimismo, pandas proporciona ciertas funcionalidades adicionales específicas de dominio, como por ejemplo la manipulación de series temporales, que no está presente en NumPy.



El cálculo orientado a arrays en Python tiene su origen en 1995, cuando Jim Hugunin creó la librería Numeric. En los siguientes diez años, muchas comunidades de programación científica empezaron a hacer programación de arrays en Python, pero el ecosistema de las librerías había empezado a fragmentarse a principios de los años 2000. En 2005, Travis Oliphant logró crear el proyecto NumPy a partir de los proyectos Numeric y Numarray de entonces, para ofrecer a la comunidad un marco único de cálculo de arrays.

Una de las razones por las que NumPy es tan importante para los cálculos numéricos en Python es porque ha sido diseñado para ser eficiente con grandes arrays de datos. Existen varias razones para esto:

- NumPy almacena internamente los datos en un bloque de memoria contiguo, independiente de otros objetos internos de Python. La librería de algoritmos de NumPy, escrita en el lenguaje C, puede funcionar en esta memoria sin comprobación de tipos u otras sobrecargas. Los arrays de NumPy utilizan además mucha menos memoria que las secuencias internas de Python.
- Las operaciones de NumPy realizan complejos cálculos sobre arrays enteros sin que haga falta usar bucles `for` de Python, proceso que puede ser lento con secuencias grandes. NumPy es más rápido que el código normal de Python, porque sus algoritmos basados en C evitan la sobrecarga que suele estar presente en el código habitual interpretado de Python.

Para dar una idea de la diferencia en rendimiento, supongamos un array NumPy de un millón de enteros, y la lista de Python equivalente:

```
In [7]: import numpy as np
```

```
In [8]: my_arr = np.arange(1_000_000)
```

```
In [9]: my_list = list(range(1_000_000))
```

Ahora multipliquemos cada secuencia por 2:

```
In [10]: %timeit my_arr2 = my_arr * 2
715 us +- 13.2 us per loop (mean +- std. dev. of 7 runs,
1000 loops each)
```

```
In [11]: %timeit my_list2 = [x * 2 for x in my_list]
48.8 ms +- 298 us per loop (mean +- std. dev. of 7 runs, 10
loops each)
```

Generalmente, los algoritmos basados en NumPy suelen ser entre 10 y 100 veces más rápidos (o más) que sus equivalentes puros de Python, y

utilizan bastante menos memoria.

4.1 El ndarray de NumPy: un objeto array multidimensional

Una de las características fundamentales de NumPy es su objeto array n-dimensional, o ndarray, un contenedor rápido y flexible para grandes conjuntos de datos en Python. Los arrays permiten realizar operaciones matemáticas con bloques de datos enteros, utilizando una sintaxis similar a las operaciones equivalentes entre elementos escalares.

Para dar una idea de cómo NumPy permite realizar cálculos en lote con una sintaxis similar a la de los valores escalares en objetos internos de Python, primero vamos a importar NumPy y a crear un pequeño array:

```
In [12]: import numpy as np
```

```
In [13]: data = np.array([[1.5, -0.1, 3], [0, -3, 6.5]])
```

```
In [14]: data
```

```
Out[14]:
```

```
array([[ 1.5, -0.1,  3. ],
       [ 0. , -3. ,  6.5]])
```

Después escribimos operaciones matemáticas con datos:

```
In [15]: data * 10
```

```
Out[15]:
```

```
array([[ 15., -1.,  30.],
       [ 0., -30.,  65.]])
```

```
In [16]: data + data
```

```
Out[16]:
```

```
array([[ 3. , -0.2,  6. ],
       [ 0. , -6. , 13. ]])
```

En el primer ejemplo, todos los elementos se han multiplicado por 10. En el segundo, los valores correspondientes de cada «celda» del array se han sumado uno con otro.



En este capítulo y a lo largo de todo el libro, emplearé el convenio estándar de NumPy de usar siempre `import numpy as np`. Sería posible poner `from numpy import *` en el código para evitar tener que escribir `np.`, pero no recomiendo acostumbrarse a esto. El espacio de nombres `numpy` es grande y contiene una serie de funciones cuyos nombres entran en conflicto con las funciones internas de Python (como `min` y `max`). Seguir convenios estándares como estos es casi siempre una buena idea.

Un `ndarray` es un contenedor genérico multidimensional para datos homogéneos; es decir, todos los elementos deben ser del mismo tipo. Cada array tiene un `shape`, una tupla que indica el tamaño de cada dimensión, y un `dtype`, un objeto que describe el tipo de datos del array:

```
In [17]: data.shape
Out[17]: (2, 3)

In [18]: data.dtype
Out[18]: dtype('float64')
```

Este capítulo presenta los fundamentos del uso de arrays NumPy, y debería bastar para poder seguir el resto del libro. Aunque no es necesario tener un profundo conocimiento de NumPy para muchas aplicaciones analíticas de datos, dominar la programación y el pensamiento orientados a arrays es un paso clave para convertirse en un gurú científico de Python.



Siempre que vea «array», «array NumPy» o «ndarray» en el texto del libro, en la mayoría de los casos el término se está refiriendo al objeto `ndarray`.

Creando ndarrays

La forma más sencilla de crear un array es mediante la función `array`. Esta función acepta cualquier objeto similar a una secuencia (incluyendo otros arrays) y produce un nuevo array NumPy conteniendo los datos pasados. Por ejemplo, una lista es una buena candidata para la conversión:

```
In [19]: data1 = [6, 7.5, 8, 0, 1]

In [20]: arr1 = np.array(data1)
```

```
In [21]: arr1
Out[21]: array([6. , 7.5, 8. , 0. , 1. ])
```

Las secuencias anidadas, como por ejemplo una lista de listas de la misma longitud, serán convertidas en un array multidimensional:

```
In [22]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
```

```
In [23]: arr2 = np.array(data2)
```

```
In [24]:                                arr2
Out[24]:
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8]])
```

Como data2 era una lista de listas, el array NumPy arr2 tiene dos dimensiones, con la forma inferida a partir de los datos. Podemos confirmar esto inspeccionando los atributos ndim y shape:

```
In [25]: arr2.ndim
Out[25]: 2
```

```
In [26]: arr2.shape
Out[26]: (2, 4)
```

A menos que se especifique de forma explícita (lo que se trata en la sección siguiente «Tipos de datos para ndarrays»), `numpy.array` trata de deducir un tipo de datos bueno para el array que crea. Dicho tipo de datos se almacena en un objeto de metadatos `dtype` especial; en los dos ejemplos anteriores tenemos:

```
In [27]: arr1.dtype
Out[27]: dtype('float64')
```

```
In [28]: arr2.dtype
Out[28]: dtype('int64')
```

Además de `numpy.array`, hay una serie de funciones adicionales para crear nuevos arrays. A modo de ejemplo, `numpy.zeros` y `numpy.ones` crean

arrays de ceros y unos, respectivamente, con una determinada longitud o forma; `numpy.empty` crea un array sin inicializar sus valores a ningún valor especial. Para crear un array de las máximas dimensiones con estos métodos, pasamos una tupla para la forma:

```
In [29]: np.zeros(10)
Out[29]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
In [30]: np.zeros((3, 6))
Out[30]:
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
In [31]: np.empty((2, 3, 2))
Out[31]:
array([[[0., 0.],
        [0., 0.],
        [0., 0.]],
       [[0., 0.],
        [0., 0.],
        [0., 0.]])
```



No es seguro suponer que `numpy.empty` devolverá un array de todo ceros. Esta función devuelve memoria no inicializada y por lo tanto puede contener valores «basura» que no son cero. Solo se debería utilizar esta función si la intención es poblar el nuevo array con datos.

`numpy.arange` es una versión con valores de array de la función interna `range` de Python:

```
In [32]: np.arange(15)
Out[32]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Consulte en la tabla 4.1 una lista de funciones estándares de creación de arrays. Como NumPy se centra en la computación numérica, el tipo de datos, si no se especifica, será en muchos casos `float64` (punto flotante).

Tabla 4.1. Algunas funciones importantes de creación de arrays NumPy.

Función	Descripción
<code>array</code>	Convierte datos de entrada (lista, tupla, array u otro tipo de secuencia) en un <code>ndarray</code> o bien deduciendo un tipo de datos o especificándolo de forma explícita; copia los datos de entrada por omisión.
<code>asarray</code>	Convierte la entrada en <code>ndarray</code> , pero no copia si la entrada ya es un <code>ndarray</code> .
<code>arange</code>	Igual que la función <code>range</code> interna, pero devuelve un <code>ndarray</code> en lugar de una lista.
<code>ones</code> , <code>ones_like</code>	Produce un array de todos unos con la forma y el tipo de datos dados; <code>ones_like</code> toma otro array y produce un array <code>ones</code> con la misma forma y tipo de datos.
<code>zeros</code> , <code>zeros_like</code>	Igual que <code>ones</code> y <code>ones_like</code> , pero produciendo arrays de ceros.
<code>empty</code> , <code>empty_like</code>	Crea nuevos arrays asignando nueva memoria, pero no los llena con valores como <code>ones</code> y <code>zeros</code> .
<code>full</code> , <code>full_like</code>	Produce un array con la forma y el tipo de datos dados y con todos los valores fijados en el «valor de relleno» indicado; <code>full_like</code> toma otro array y produce un array relleno con la misma forma y tipo de datos.
<code>eye</code> , <code>identity</code>	Crea una matriz de identidad cuadrada $N \times N$ (con unos en la diagonal y ceros en el resto).

Tipos de datos para `ndarrays`

El tipo de datos o `dtype` es un objeto especial que contiene la información (o los metadatos, datos sobre datos) que el `ndarray` necesita para interpretar un fragmento de memoria como un determinado tipo de datos:

```
In [2]: tup = (4, 5, 6)
```

```
In [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)
```

```
In [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)
```

```
In [35]: arr1.dtype
Out[35]: dtype('float64')
```

```
In [36]: arr2.dtype
Out[36]: dtype('int32')
```

Los tipos de datos son una fuente de flexibilidad de NumPy para interactuar con datos procedentes de otros sistemas. En la mayoría de los casos ofrecen un mapeado directamente sobre la representación subyacente de un disco o memoria, lo que hace posible leer y escribir flujos de datos binarios en disco y conectar con código escrito en un lenguaje de bajo nivel como C o FORTRAN. Los tipos de datos numéricos se denominan de la misma manera: un nombre de tipo, como `float` o `int`, seguido de un número que indica el número de bits por elemento. Un valor estándar de punto flotante y doble precisión (que se ha usado internamente en el objeto `float` de Python) requiere hasta 8 bytes o 64 bits. Así, este tipo se conoce en NumPy como `float64`. Véase en la tabla 4.2 el listado completo de los tipos de datos soportados por NumPy.


 No hay que preocuparse por memorizar los tipos de datos de NumPy, especialmente en el caso de los nuevos usuarios. A menudo solo basta con tener en cuenta el tipo de datos general que se está manejando, ya sea punto flotante, complejo, entero, booleano, cadena de texto o un objeto general de Python. Cuando sea necesario tener más control sobre el modo en que los datos se almacenan en memoria y en disco, especialmente con grandes conjuntos de datos, es bueno saber que se tiene control sobre el tipo de almacenamiento.

Tabla 4.2. Tipos de datos de NumPy.

Tipo	Código del tipo	Descripción
<code>int8, uint8</code>	<code>i1, u1</code>	Tipos enteros con y sin signo de 8 bits (1 byte).
<code>int16, uint16</code>	<code>i2, u2</code>	Tipos enteros con y sin signo de 16 bits.
<code>int32, uint32</code>	<code>i4, u4</code>	Tipos enteros con y sin signo de 32 bits.
<code>int64, uint64</code>	<code>i8, u8</code>	Tipos enteros con y sin signo de 64 bits.
<code>float16</code>	<code>f2</code>	Punto flotante de precisión media.
<code>float32</code>	<code>f4</code> o <code>f</code>	Punto flotante estándar de precisión sencilla; compatible con <code>float</code> de C.
<code>float64</code>	<code>f8</code> o <code>d</code>	Punto flotante estándar de precisión doble; compatible con <code>double</code> de C y el objeto <code>float</code> de Python.

float128	f16 o g	Punto flotante de precisión extendida.
complex64, complex128, complex256	c8, c16, c32	Números complejos representados por dos floats de 32, 64 y 120, respectivamente.
bool	?	Tipo booleano que almacena valores True y False.
object	0	Tipo de objeto Python; un valor puede ser cualquier objeto Python.
string_	s	Tipo de cadena de texto ASCII de longitud fija (1 byte por carácter); por ejemplo, para crear un tipo de datos de cadena de texto con longitud 10, utilizamos "s10".
unicode_	u	Tipo Unicode de longitud fija (el número de bytes es específico de la plataforma); la misma semántica de especificación que string_ (por ejemplo, "u10").



Existen tipos enteros con y sin signo, y quizá muchos lectores no estén familiarizados con esta terminología. Un entero con signo puede representar enteros positivos y negativos, mientras que un entero sin signo solo puede representar enteros que no sean cero. Por ejemplo, `int8` (entero con signo de 8 bits) puede representar enteros de -128 a 127 (inclusive), mientras que `uint8` (entero sin signo de 8 bits) puede representar de 0 a 255.

Se puede convertir de forma explícita un array de un tipo de datos a otro utilizando el método `astype` de `ndarray`:

```
In [37]: arr = np.array([1, 2, 3, 4, 5])

In [38]: arr.dtype
Out[38]: dtype('int64')

In [39]: float_arr = arr.astype(np.float64)

In [40]: float_arr
Out[40]: array([1., 2., 3., 4., 5.])

In [41]: float_arr.dtype
Out[41]: dtype('float64')
```

En este ejemplo, los enteros fueron convertidos a punto flotante. Si se convierten números en punto flotante al tipo de datos entero, la parte decimal quedará truncada:

```
In [42]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
```

```
In [43]: arr
```

```
Out[43]: array([ 3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
```

```
In [44]: arr.astype(np.int32)
```

```
Out[44]: array([ 3, -1, -2, 0, 12, 10], dtype=int32)
```

Si tenemos un array de cadenas de texto que representan números, se puede emplear `astype` para convertirlos a su forma numérica:

```
In [45]: numeric_strings = np.array(["1.25", "-9.6", "42"],  
dtype=np.string_)
```

```
In [46]: numeric_strings.astype(float)
```

```
Out[46]: array([ 1.25, -9.6 , 42. ])
```



Conviene tener cuidado al utilizar el tipo `numpy.string_`, ya que los datos de cadena de texto de NumPy tienen tamaño fijo y la entrada puede quedar truncada sin previo aviso. `pandas` tiene un comportamiento más intuitivo con datos no numéricos.

Si la conversión fallara por alguna razón (como, por ejemplo, que una cadena de texto no se pudiera convertir a `float64`), se produciría un `ValueError`. Antes solía ser algo perezoso y escribía `float` en lugar de `np.float64`; NumPy asigna a los tipos de Python sus propios tipos de datos equivalentes.

También se puede emplear el atributo `dtype` de otro array:

```
In [47]: int_array = np.arange(10)
```

```
In [48]: calibers = np.array([.22, .270, .357, .380, .44,  
.50], dtype=np.float64)
```

```
In [49]: int_array.astype(calibers.dtype)
```

```
Out[49]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

Existen cadenas de texto para código abreviadas que se pueden usar también para referirse a un `dtype`:

```
In [50]: zeros_uint32 = np.zeros(8, dtype="u4")
```

```
In [51]: zeros_uint32
```

```
Out[51]: array([0, 0, 0, 0, 0, 0, 0, 0], dtype=uint32)
```



Llamar a `astype` crea siempre un nuevo array (una copia de los datos), incluso aunque el nuevo tipo de datos sea el mismo que el antiguo.

Aritmética con arrays NumPy

Los arrays son importantes porque permiten expresar operaciones en lotes con datos sin escribir bucles `for`. Los usuarios de NumPy llaman a esto vectorización. Cualquier operación aritmética entre arrays del mismo tamaño se aplica elemento por elemento:

```
In [52]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])
```

```
In [53]: arr
```

```
Out[53]:
```

```
array([[1., 2., 3.],  
       [4., 5., 6.]])
```

```
In [54]: arr * arr
```

```
Out[54]:
```

```
array([[ 1.,  4.,  9.],  
       [16., 25., 36.]])
```

```
In [55]: arr - arr
```

```
Out[55]:
```

```
array([[0., 0., 0.],  
       [0., 0., 0.]])
```

Las operaciones aritméticas con escalares propagan el argumento escalar a cada elemento del array:

```
In [56]: 1 / arr
```

```
Out[56]:
```

```
array([[1. , 0.5 , 0.3333],  
       [0.25 , 0.2 , 0.1667]])
```



```
In [57]: arr ** 2
Out[57]:
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])
```

Las comparaciones entre arrays del mismo tamaño producen arrays booleanos:

```
In [58]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
```

```
In [59]: arr2
Out[59]:
array([[ 0.,  4.,  1.],
       [ 7.,  2., 12.]])
```

```
In [60]: arr2 > arr
Out[60]:
array([[False,  True, False],
       [ True, False,  True]])
```

Al proceso de evaluar operaciones entre arrays de distintos tamaños se le denomina difusión, que trataremos posteriormente en el apéndice A. Tener un conocimiento profundo de la difusión no es necesario para la mayor parte de este libro.

Indexado y corte básicos

El indexado de array de NumPy es un tema de gran profundidad, ya que hay muchas formas en las que se puede querer seleccionar un subconjunto de datos o elementos individuales. Los arrays unidimensionales son sencillos; a primera vista actúan de forma similar a las listas de Python:

```
In [61]: arr = np.arange(10)
```

```
In [62]: arr
Out[62]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [63]: arr[5]
Out[63]: 5
```

```
In [64]: arr[5:8]
Out[64]: array([5, 6, 7])
```

```
In [65]: arr[5:8] = 12
```

```
In [66]: arr
Out[66]: array([ 0, 1, 2, 3, 4, 12, 12, 12, 8, 9])
```

Como hemos visto, si se asigna un valor escalar a un corte, como en `arr[5:8] = 12`, el valor se propaga (o difunde) a toda la selección.



Una importante primera distinción en las listas internas de Python es que los cortes de array son vistas del array original, lo que significa que los datos no se copian, y que cualquier modificación de la vista se verá reflejada en el array de origen.

Para dar un ejemplo de esto, primero creamos un corte de `arr`:

```
In [67]: arr_slice = arr[5:8]
```

```
In [68]: arr_slice
Out[68]: array([12, 12, 12])
```

A continuación, cuando se cambian los valores de `arr_slice`, las mutaciones se reflejan en el array original `arr`:

```
In [69]: arr_slice[1] = 12345
```

```
In [70]: arr
Out[70]:
array([ 0, 1, 2, 3, 4, 12, 12345, 12, 8, 9])
```

El corte «vacío» `[:]` se asignará a todos los valores del array:

```
In [71]: arr_slice[:] = 64
```

```
In [72]: arr
Out[72]: array([ 0, 1, 2, 3, 4, 64, 64, 64, 8, 9])
```

A los principiantes en NumPy quizá esto les sorprenda, especialmente si han utilizado otros lenguajes de programación de array que copian datos con más empeño. Como NumPy ha sido diseñado para poder trabajar con

arrays muy grandes, podríamos esperar problemas de rendimiento y memoria si NumPy insistiera en copiar siempre datos.



Si nos interesa más una copia de un corte de un ndarray que una vista, tendremos que copiar explícitamente el array (por ejemplo, `arr[5:8].copy()`). Como veremos más tarde, pandas funciona también de este modo.

Con arrays de muchas dimensiones tenemos muchas más opciones. En un array bidimensional, los elementos de cada índice ya no son escalares, sino más bien arrays unidimensionales:

```
In [73]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  
  
In [74]: arr2d[2]  
Out[74]: array([7, 8, 9])
```

Por lo tanto, se puede acceder los elementos individuales de forma recursiva. Pero esto es demasiado trabajo, así que lo que se puede hacer es pasar una lista de índices separados por comas para seleccionar elementos individuales. Entonces estas expresiones son equivalentes:

```
In [75]: arr2d[0][2]  
Out[75]: 3  
  
In [76]: arr2d[0, 2]  
Out[76]: 3
```

Véase en la figura 4.1 una ilustración del indexado en un array bidimensional. Resulta útil pensar en el eje 0 como en las «filas» del array y en el eje 1 como en las «columnas».

		Eje 1		
		0	1	2
Eje 0	0	0,0	0,1	0,2
	1	1,0	1,1	1,2
	2	2,0	2,1	2,2

Figura 4.1. Indexado de elementos en un array NumPy.

En arrays multidimensionales, si se omiten los índices posteriores, el objeto devuelto será un ndarray de menos dimensiones formado por todos los datos de las dimensiones superiores. De forma que en el array `arr3d` de $2 \times 2 \times 3$:

```
In [77]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

```
In [78]: arr3d
Out[78]:
array([[[ 1, 2, 3],
        [ 4, 5, 6]],
       [[ 7, 8, 9],
        [10, 11, 12]]])
```

`arr3d[0]` es un array de 2×3 :

```
In [79]: arr3d[0]
Out[79]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Se pueden asignar tanto valores escalares como arrays a `arr3d[0]`:

```
In [80]: old_values = arr3d[0].copy()
```

```
In [81]: arr3d[0] = 42
```

```
In [82]: arr3d
Out[82]:
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

```
In [83]: arr3d[0] = old_values
```

```
In [84]: arr3d
Out[84]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

De forma similar, `arr3d[1, 0]` da todos los valores cuyos índices empiezan por (1, 0), formando un array unidimensional:

```
In [85]: arr3d[1, 0]
Out[85]: array([7, 8, 9])
```

Esta expresión es la misma que si hubiéramos indexado en dos pasos:

```
In [86]: x = arr3d[1]
```

```
In [87]: x
Out[87]:
array([[ 7,  8,  9],
       [10, 11, 12]])
```

```
In [88]: x[0]
Out[88]: array([7, 8, 9])
```

Hay que tener en cuenta que, en todos estos casos en los que se han seleccionado subsecciones del array, los arrays devueltos son vistas.



Esta sintaxis de indexado multidimensional para arrays NumPy no funcionará con objetos Python normales, como por ejemplo listas de listas.

Indexar con cortes

Al igual que los objetos unidimensionales como las listas de Python, los ndarrays se pueden cortar con la sintaxis habitual:

```
In [89]: arr
Out[89]: array([ 0, 1, 2, 3, 4, 64, 64, 64, 8, 9])

In [90]: arr[1:6]
Out[90]: array([ 1, 2, 3, 4, 64])
```

Veamos el array bidimensional de antes, arr2d. Cortar este array es algo distinto:

```
In [91]: arr2d
Out[91]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [92]: arr2d[:2]
Out[92]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Como se puede ver, se ha cortado a lo largo del eje 0, el primer eje. Por lo tanto, un corte selecciona un rango de elementos a lo largo de un eje. Puede resultar útil leer la expresión `arr2d[:2]` como «seleccionar las primeras dos filas de arr2d».

Se pueden pasar varios cortes igual que se pasan varios índices:

```
In [93]: arr2d[:2, 1:]
Out[93]:
```

```
array([[2, 3],  
       [5, 6]])
```

Al realizar así los cortes, siempre se obtienen vistas de array del mismo número de dimensiones. Mezclando índices enteros y cortes, se obtienen cortes de menos dimensiones.

Por ejemplo, podemos seleccionar la segunda fila pero solo las primeras dos columnas, de esta forma:

```
In [94]: lower_dim_slice = arr2d[1, :2]
```

Aquí, aunque `arr2d` es bidimensional, `lower_dim_slice` es unidimensional, y su forma es una tupla con un solo tamaño de eje:

```
In [95]: lower_dim_slice.shape  
Out[95]: (2,)
```

De forma similar, es posible elegir la tercera columna pero solo las dos primeras filas, de este modo:

```
In [96]: arr2d[:2, 2]  
Out[96]: array([3, 6])
```

Véase la ilustración de la figura 4.2. Un signo de punto y coma por sí solo significa tomar el eje entero, así que se pueden cortar solo ejes de mayor dimensión haciendo lo siguiente:

```
In [97]: arr2d[:, :1]  
Out[97]:  
array([[1],  
       [4],  
       [7]])
```

Por supuesto, asignar a una expresión de corte asigna a la selección completa:

```
In [98]: arr2d[:2, 1:] = 0
```

```
In [99]: arr2d
Out[99]:
array([[1, 0, 0],
       [4, 0, 0],
       [7, 8, 9]])
```

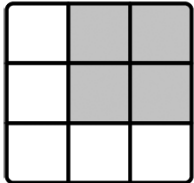
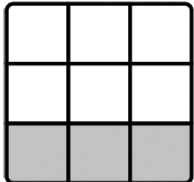
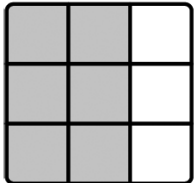
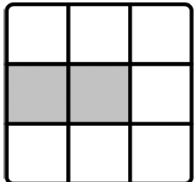
	Expresión	Forma
	<code>arr[:2,1:]</code>	<code>(2,2)</code>
	<code>arr[2]</code>	<code>(3,)</code>
	<code>arr[2, :]</code>	<code>(3,)</code>
	<code>arr[2:, :]</code>	<code>(1,3)</code>
	<code>arr[:, :2]</code>	<code>(3,2)</code>
	<code>arr[1, :2]</code>	<code>(2,)</code>
	<code>arr[1:2, :2]</code>	<code>(1,2)</code>

Figura 4.2. Corte de array bidimensional.

Indexado booleano

Veamos un ejemplo en el que tenemos datos en un array y un array de nombres con duplicados:

```
In [100]: names = np.array(["Bob", "Joe", "Will", "Bob",
                             "Will", "Joe", "Joe"])
In [101]: data = np.array([[4, 7], [0, 2], [-5, 6], [0, 0],
                             [1, 2],
```



```
.....:          [-12, -4], [3, 4]])
```

```
In [102]: names
```

```
Out[102]: array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe',  
'Joe'], dtype='<U4')
```

```
In [103]: data
```

```
Out[103]:  
array([[ 4,  7],  
       [ 0,  2],  
       [-5,  6],  
       [ 0,  0],  
       [ 1,  2],  
       [-12, -4],  
       [ 3,  4]])
```

Supongamos que cada nombre corresponde a una fila del array de datos y que queremos seleccionar todas las filas con el nombre “Bob”. Como las operaciones aritméticas, las comparaciones con arrays (como por ejemplo `==`) también son vectorizadas. Así, comparar nombres con la cadena de texto “Bob” produce un array booleano:

```
In [104]: names == “Bob”
```

```
Out[104]: array([ True, False, False,  True, False, False,  
False])
```

Este array booleano puede pasarse al indexar el array:

```
In [105]: data[names == “Bob”]
```

```
Out[105]:  
array([[4, 7],  
       [0, 0]])
```

El array booleano debe tener la misma longitud que el eje del array que está indexando. Incluso se pueden mezclar y combinar arrays booleanos con cortes o enteros (o secuencias de enteros; veremos más después en este mismo capítulo).

En estos ejemplos, seleccionamos de las filas en las que `names == "Bob"` e indexamos también las columnas:

```
In [106]: data[names == "Bob", 1:]  
Out[106]:  
array([[7],  
       [0]])
```

```
In [107]: data[names == "Bob", 1]  
Out[107]: array([7, 0])
```

Para seleccionar todo excepto "Bob", se puede utilizar `!=` o negar la condición colocando delante el operador `~`:

```
In [108]: names != "Bob"  
Out[108]: array([False,  True,  True,  False,  True,  True,  
                True])
```

```
In [109]: ~(names == "Bob")  
Out[109]: array([False,  True,  True,  False,  True,  True,  
                True])
```

```
In [110]: data[~(names == "Bob")]  
Out[110]:  
array([[ 0,  2],  
       [-5,  6],  
       [ 1,  2],  
       [-12, -4],  
       [ 3,  4]])
```

El operador `~` puede ser útil cuando se desea invertir un array booleano al que se ha hecho referencia con una variable:

```
In [111]: cond = names == "Bob"
```

```
In [112]: data[~cond]  
Out[112]:  
array([[ 0,  2],  
       [-5,  6],  
       [ 1,  2],
```

```
[-12, -4],  
[ 3, 4]])
```

Para elegir dos de los tres nombres y combinar así varias condiciones booleanas, utilizamos operadores aritméticos booleanos como & (and) y | (or):

```
In [113]: mask = (names == "Bob") | (names == "Will")
```

```
In [114]: mask
```

```
Out[114]: array([ True, False,  True,  True,  True, False,  
                False])
```

```
In [115]: data[mask]
```

```
Out[115]:
```

```
array([[ 4,  7],  
       [-5,  6],  
       [ 0,  0],  
       [ 1,  2]])
```

Seleccionar datos de un array mediante indexado booleano y asignar el resultado a una nueva variable siempre crea una copia de los datos, incluso aunque el array devuelto no haya sido modificado.



Las palabras clave de Python and y or no funcionan con arrays booleanos. Emplee en su lugar & (and) y | (or).

Configurar valores con arrays booleanos funciona sustituyendo el valor o valores del lado derecho en las ubicaciones en las que los valores del array booleano son True. Para configurar todos los valores negativos de los datos a 0, solo necesitamos hacer esto:

```
In [116]: data[data < 0] = 0
```

```
In [117]: data
Out[117]:
array([[4, 7],
       [0, 2],
       [0, 6],
       [0, 0],
       [1, 2],
       [0, 0],
       [3, 4]])
```

También se pueden configurar filas o columnas completas utilizando un array booleano unidimensional:

```
In [118]: data[names != "Joe"] = 7
```

```
In [119]: data
Out[119]:
array([[7, 7],
       [0, 2],
       [7, 7],
       [7, 7],
       [7, 7],
       [0, 0],
       [3, 4]])
```

Como veremos después, estos tipos de operaciones en datos bidimensionales son cómodas de realizar con pandas.

Indexado sofisticado

El indexado sofisticado es un término adoptado por NumPy para describir el indexado utilizando arrays enteros. Supongamos que tenemos un array de 8×4 :

```
In [120]: arr = np.zeros((8, 4))
```

```
In [121]: for i in range(8):
.....:         arr[i] = i
```

```
In [122]: arr
Out[122]:
array([[0., 0., 0., 0.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [3., 3., 3., 3.],
       [4., 4., 4., 4.],
       [5., 5., 5., 5.],
       [6., 6., 6., 6.],
       [7., 7., 7., 7.]])
```

Para seleccionar un subconjunto de las filas en un determinado orden, se puede simplemente pasar una lista o ndarray de enteros especificando el orden deseado:

```
In [123]: arr[[4, 3, 0, 6]]
Out[123]:
array([[4., 4., 4., 4.],
       [3., 3., 3., 3.],
       [0., 0., 0., 0.],
       [6., 6., 6., 6.]])
```

Supuestamente este código hizo lo esperado. Utilizar índices negativos selecciona filas desde el final:

```
In [124]: arr[[-3, -5, -7]]
Out[124]:
array([[5., 5., 5., 5.],
       [3., 3., 3., 3.],
       [1., 1., 1., 1.]])
```

Pasar varios arrays de índice hace algo un poco distinto; selecciona un array unidimensional de elementos correspondiente a cada tupla de índices:

```
In [125]: arr = np.arange(32).reshape((8, 4))
```

```
In [126]: arr
Out[126]:
array([[ 0,  1,  2,  3],
```

```
[ 4, 5, 6, 7],  
[ 8, 9, 10, 11],  
[12, 13, 14, 15],  
[16, 17, 18, 19],  
[20, 21, 22, 23],  
[24, 25, 26, 27],  
[28, 29, 30, 31]])
```

```
In [127]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]  
Out[127]: array([ 4, 23, 29, 10])
```

Para saber más del método reshape, eche un vistazo al apéndice A.

Aquí se seleccionaron los elementos (1, 0), (5, 3), (7, 1) y (2, 2). El resultado del indexado sofisticado con tantos arrays de enteros como ejes es siempre unidimensional.

El comportamiento del indexado sofisticado en este caso es algo distinto a lo que algunos usuarios podrían haber imaginado (yo mismo incluido), que es la región rectangular formada por elegir un subconjunto de las filas y columnas de la matriz. Esta es una forma de conseguir esto:

```
In [128]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]  
Out[128]:  
array([[ 4, 7, 5, 6],  
[20, 23, 21, 22],  
[28, 31, 29, 30],  
[ 8, 11, 9, 10]])
```

Conviene recordar que el indexado sofisticado, a diferencia del corte, copia siempre los datos en un nuevo array al asignar el resultado a una nueva variable. Si se asignan valores con indexado sofisticado, los valores indexados se modificarán:

```
In [129]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]  
Out[129]: array([ 4, 23, 29, 10])
```

```
In [130]: arr[[1, 5, 7, 2], [0, 3, 1, 2]] = 0
```

```
In [131]: arr
```

```
Out[131]:  
array([[ 0,  1,  2,  3],  
       [ 0,  5,  6,  7],  
       [ 8,  9,  0, 11],  
       [12, 13, 14, 15],  
       [16, 17, 18, 19],  
       [20, 21, 22,  0],  
       [24, 25, 26, 27],  
       [28,  0, 30, 31]])
```

Transponer arrays e intercambiar ejes

Transponer es una forma especial de remodelación que devuelve de forma similar una vista de los datos subyacentes sin copiar nada. Los arrays tienen el método `transpose` y el atributo especial `T`:

```
In [132]: arr = np.arange(15).reshape((3, 5))
```

```
In [133]: arr  
Out[133]:  
array([[ 0,  1,  2,  3,  4],  
       [ 5,  6,  7,  8,  9],  
       [10, 11, 12, 13, 14]])
```

```
In [134]: arr.T  
Out[134]:  
array([[ 0,  5, 10],  
       [ 1,  6, 11],  
       [ 2,  7, 12],  
       [ 3,  8, 13],  
       [ 4,  9, 14]])
```

Cuando se realizan cálculos con matrices, es probable que se haga esto con mucha frecuencia, por ejemplo cuando se calcula el producto interno de una matriz utilizando `numpy.dot`:

```
In [135]: arr = np.array([[0, 1, 0], [1, 2, -2], [6, 3, 2],  
                          [-1, 0, -1], [1, 0, 1]])
```

```
In [136]: arr
```

```
Out[136]:
```

```
array([[ 0,  1,  0],  
       [ 1,  2, -2],  
       [ 6,  3,  2],  
       [-1,  0, -1],  
       [ 1,  0,  1]])
```

```
In [137]: np.dot(arr.T, arr)
```

```
Out[137]:
```

```
array([[39, 20, 12],  
       [20, 14,  2],  
       [12,  2, 10]])
```

El operador @ es otra forma de hacer multiplicación de matrices:

```
In [138]: arr.T @ arr
```

```
Out[138]:
```

```
array([[39, 20, 12],  
       [20, 14,  2],  
       [12,  2, 10]])
```

La transposición sencilla con .T es un caso especial de intercambio de ejes. ndarray tiene el método swapaxes, que toma un par de números de eje e intercambia los ejes indicados para reordenar los datos:

```
In [139]: arr
```

```
Out[139]:
```

```
array([[ 0,  1,  0],  
       [ 1,  2, -2],  
       [ 6,  3,  2],  
       [-1,  0, -1],  
       [ 1,  0,  1]])
```

```
In [140]: arr.swapaxes(0, 1)
```

```
Out[140]:
```

```
array([[ 0,  1,  6, -1,  1],  
       [ 1,  2,  3,  0,  0],
```



```
[ 0, -2, 2, -1, 1]])
```

`swapaxes` devuelve de forma parecida una vista de los datos sin hacer una copia.

4.2 Generación de números pseudoaleatoria

El módulo `numpy.random` complementa el módulo `random` interno de Python con funciones para generar de forma eficaz arrays enteros de valores de muestra de muchos tipos de distribuciones de probabilidad. Por ejemplo, se puede obtener un array 4×4 de muestras de la distribución normal estándar utilizando `numpy.random.standard_normal`:

```
In [141]: samples = np.random.standard_normal(size=(4, 4))
```

```
In [142]: samples
```

```
Out[142]:
```

```
array([[ -0.2047,  0.4789, -0.5194, -0.5557],  
       [ 1.9658,  1.3934,  0.0929,  0.2817],  
       [ 0.769 ,  1.2464,  1.0072, -1.2962],  
       [ 0.275 ,  0.2289,  1.3529,  0.8864]])
```

El módulo `random` interno de Python, en contraste, solo muestrea un único valor cada vez. Como se puede ver en estos valores de referencia, `numpy.random` es más de una orden de magnitud más rápido para generar muestras muy grandes:

```
In [143]: from random import normalvariate
```

```
In [144]: N = 1_000_000
```

```
In [145]: %timeit samples = [normalvariate(0, 1) for _ in  
range(N)]
```

```
1.04 s +- 11.4 ms per loop (mean +- std. dev. of 7 runs, 1  
loop each)
```

```
In [146]: %timeit np.random.standard_normal(N)
```

21.9 ms +- 155 us per loop (mean +- std. dev. of 7 runs, 10 loops each)

Estos números aleatorios no lo son realmente (son más bien pseudoaleatorios); en realidad los produce un generador de números aleatorios configurable, que determina de forma preestablecida qué valores se crean. Funciones como `numpy.random.standard_normal` emplean el generador de números aleatorios predeterminado del módulo `numpy.random`, pero el código se puede configurar para que utilice un generador explícito:

```
In [147]: rng = np.random.default_rng(seed=12345)
```

```
In [148]: data = rng.standard_normal((2, 3))
```

El argumento `seed` determina el estado inicial del generador, pero el estado cambia cada vez que se utiliza el objeto `rng` para generar datos. El objeto de generador `rng` está también aislado de otro código que podría utilizar el módulo `numpy.random`:

```
In [149]: type(rng)
```

```
Out[149]: numpy.random._generator.Generator
```

Véase en la tabla 4.3 una lista parcial de los métodos disponibles en objetos para generación aleatoria como `rng`. Utilizaremos el objeto `rng` creado anteriormente para generar datos aleatorios en el resto del capítulo.

Tabla 4.3. Métodos generadores de números aleatorios de NumPy.

Método	Descripción
<code>permutation</code>	Devuelve una permutación aleatoria de una secuencia, o un rango permutado.
<code>shuffle</code>	Permuta aleatoriamente una secuencia en su lugar.
<code>uniform</code>	Saca muestras a partir de una distribución uniforme.
<code>integers</code>	Saca enteros aleatorios a partir de un determinado rango de menor a mayor.
<code>standard_normal</code>	Saca muestras a partir de una distribución normal con media 0 y desviación

	estándar 1.
binomial	Saca muestras a partir de una distribución binomial.
normal	Saca muestras a partir de una distribución normal (gaussiana).
beta	Saca muestras a partir de una distribución beta.
chisquare	Saca muestras a partir de una distribución chi cuadrada.
gamma	Saca muestras a partir de una distribución gamma.
uniform	Saca muestras a partir de una distribución uniforme [0, 1).

4.3 Funciones universales: funciones rápidas de array elemento a elemento

Una función universal, o *ufunc*, es una función que realiza operaciones elemento a elemento en ndarrays. Se puede pensar en ellas como si fueran rápidos contenedores vectorizados para funciones sencillas, que toman uno o varios valores escalares y producen uno o varios resultados escalares.

Muchas *ufuncs* son sencillas transformaciones elemento a elemento, como `numpy.sqrt` o `numpy.exp`:

```
In [150]: arr = np.arange(10)
```

```
In [151]: arr
```

```
Out[151]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [152]: np.sqrt(arr)
```

```
Out[152]:
```

```
array([0. , 1. , 1.4142, 1.7321, 2. , 2.2361, 2.4495,
2.6458, 2.8284, 3. ])
```

```
In [153]: np.exp(arr)
```

```
Out[153]:
```

```
array([ 1. , 2.7183, 7.3891, 2 0.0855, 54.5982,
148.4132,
403.4288, 1096.6332, 2980.958 , 8103.0839])
```

Estas funciones se denominan *ufuncs* unarias. Otras, como `numpy.add` o `numpy.maximum`, toman dos arrays (de ahí que se llamen *ufuncs* binarias) y devuelven un array sencillo como resultado:

```
In [154]: x = rng.standard_normal(8)
```

```
In [155]: y = rng.standard_normal(8)
```

```
In [156]: x
```

```
Out[156]:
```

```
array([-1.3678,  0.6489,  0.3611, -1.9529,  2.3474,  0.9685,  
       -0.7594,  0.9022])
```

```
In [157]: y
```

```
Out[157]:
```

```
array([-0.467 , -0.0607,  0.7888, -1.2567,  0.5759,  1.399 ,  
       1.3223, -0.2997])
```

```
In [158]: np.maximum(x, y)
```

```
Out[158]:
```

```
array([-0.467 ,  0.6489,  0.7888, -1.2567,  2.3474,  1.399 ,  
       1.3223,  0.9022])
```

En este ejemplo, `numpy.maximum` calculaba el máximo elemento a elemento de los elementos de `x` e `y`.

Aunque no es habitual, una *ufunc* puede devolver varios arrays. `numpy.modf` es un ejemplo: una versión vectorizada de la función `math.modf` interna de Python, devuelve las partes fraccionaria y entera de un array de punto flotante:

```
In [159]: arr = rng.standard_normal(7) * 5
```

```
In [160]: arr
```

```
Out[160]: array([ 4.5146, -8.1079, -0.7909,  2.2474, -6.718 ,  
                -0.4084,  8.6237])
```

```
In [161]: remainder, whole_part = np.modf(arr)
```

```
In [162]: remainder
```

```
Out[162]: array([ 0.5146, -0.1079, -0.7909,  0.2474, -0.718 ,  
                -0.4084,  0.6237])
```

```
In [163]: whole_part
Out[163]: array([ 4., -8., -0., 2., -6., -0., 8.] )
```

Las *ufuncs* aceptan un argumento opcional *out*, que les permite asignar sus resultados a un array existente en lugar de crear uno nuevo:

```
In [164]: arr
Out[164]: array([ 4.5146, -8.1079, -0.7909, 2.2474, -6.718 ,
-0.4084, 8.6237])
```

```
In [165]: out = np.zeros_like(arr)
```

```
In [166]: np.add(arr, 1)
Out[166]: array([ 5.5146, -7.1079, 0.2091, 3.2474, -5.718 ,
0.5916, 9.6237])
```

```
In [167]: np.add(arr, 1, out=out)
Out[167]: array([ 5.5146, -7.1079, 0.2091, 3.2474, -5.718 ,
0.5916, 9.6237])
```

```
In [168]: out
Out[168]: array([ 5.5146, -7.1079, 0.2091, 3.2474, -5.718 ,
0.5916, 9.6237])
```

Véanse en las tablas 4.4 y 4.5 algunas de las *ufuncs* de NumPy. Constantemente se añaden nuevas funciones de este tipo a NumPy, por lo tanto consultar su documentación en línea es la mejor forma de disponer de un listado completo y totalmente actualizado.

Tabla 4.4. Algunas funciones universales unarias.

Función	Descripción
abs, fabs	Calcula el valor absoluto elemento a elemento para valores enteros, de punto flotante o complejos.
sqrt	Calcula la raíz cuadrada de cada elemento (equivalente a <code>arr ** 0.5</code>).
square	Calcula el cuadrado de cada elemento (equivalente a <code>arr ** 2</code>).
exp	Calcula el exponente e^x de cada elemento.
log, log10, log2, log1p	Logaritmo natural (base e), logaritmo en base 10, logaritmo en

	base 2 y $\log(1 + x)$, respectivamente.
sign	Calcula el signo de cada elemento: 1 (positivo), 0 (cero) o -1 (negativo).
ceil	Calcula el valor máximo de cada elemento (es decir, el entero más pequeño mayor o igual a él).
floor	Calcula el valor mínimo de cada elemento (es decir, el entero más grande menor o igual a él).
rint	Redondea los elementos al entero más próximo, preservando el dtype.
modf	Devuelve las partes fraccionaria y entera de un array como arrays distintos.
isnan	Devuelve un valor booleano indicando si cada valor es NaN (Not a Number: no es un número).
isfinite, isinf	Devuelve un array booleano indicando si cada elemento es finito (no inf, no NaN) o infinito, respectivamente.
cos, cosh, sin, sinh, tan, tanh	Funciones trigonométricas normales e hiperbólicas.
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	Funciones trigonométricas inversas.
logical_not	Calcula el valor de verdad de not x elemento a elemento (equivalente a $\sim arr$).

Tabla 4.5. Algunas funciones universales binarias.

Función	Descripción
add	Suma los elementos correspondientes de varios arrays.
subtract	Resta al primer array los elementos del segundo.
multiply	Multiplica los elementos de un array.
divide, floor_divide	Divide o aplica la división de piso (truncando el resto).
power	Eleva los elementos del primer array a las potencias indicadas en el segundo.
maximum, fmax	Máximo elemento a elemento; fmax ignora NaN.

Función	Descripción
<code>minimum, fmin</code>	Mínimo elemento a elemento; <code>fmin</code> ignora NaN.
<code>mod</code>	Módulo elemento a elemento (resto de la división).
<code>copysign</code>	Copia el signo de los valores del segundo argumento al primero.
<code>greater, greater_equal, less, less_equal, equal, not_equal</code>	Realiza comparaciones elemento a elemento, produciendo un array booleano (equivalente a las operaciones <code>></code> , <code>>=</code> , <code><</code> , <code><=</code> , <code>==</code> , <code>!=</code>).
<code>logical_and</code>	Calcula elemento a elemento el valor de verdad de la operación lógica AND (<code>&</code>).
<code>logical_or</code>	Calcula elemento a elemento el valor de verdad de la operación lógica OR (<code> </code>).
<code>logical_xor</code>	Calcula elemento a elemento el valor de verdad de la operación lógica XOR (<code>^</code>).

4.4 Programación orientada a arrays con arrays

El uso de arrays NumPy permite expresar muchos tipos de procesos de datos como concisos arrays que, de otro modo, podrían requerir la escritura de bucles. Algunos denominan vectorización a esta práctica de reemplazar bucles explícitos por expresiones array. En general, las operaciones array vectorizadas suelen ser bastante más rápidas que sus equivalentes puros de Python y tienen un máximo impacto en cualquier tipo de cálculo numérico. Más adelante, en el apéndice A, explicaré la difusión, un potente método para vectorizar cálculos.

Como ejemplo sencillo, supongamos que queremos evaluar la función $\sqrt{x^2 + y^2}$ a lo largo de una cuadrícula de valores. La función `numpy.meshgrid` toma dos arrays unidimensionales y produce dos matrices bidimensionales que corresponden a todos los pares de (x, y) de los dos arrays:

```
In [169]: points = np.arange(-5, 5, 0.01) # 100 puntos
          espaciados por igual
```

```
In [170]: xs, ys = np.meshgrid(points, points)
```

```
In [171]: ys
```

```
Out[171]:
```

```
array([[ -5. ,  -5. ,  -5. , ...,  -5. ,  -5. ,  -5. ],  
       [ -4.99,  -4.99,  -4.99, ...,  -4.99,  -4.99,  -4.99],  
       [ -4.98,  -4.98,  -4.98, ...,  -4.98,  -4.98,  -4.98],  
       ...,  
       [  4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],  
       [  4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],  
       [  4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

Ahora, evaluar la función es cuestión de escribir la misma expresión que escribiríamos con dos puntos:

```
In [172]: z = np.sqrt(xs ** 2 + ys ** 2)
```

```
In [173]: z
```

```
Out[173]:
```

```
array([[7.0711,  7.064 ,  7.0569, ...,  7.0499,  7.0569,  7.064  
       ],  
       [7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0499,  7.0569],  
       [7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0428,  7.0499],  
       ...,  
       [7.0499,  7.0428,  7.0357, ...,  7.0286,  7.0357,  7.0428],  
       [7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0428,  7.0499],  
       [7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0499,  7.0569]])
```

A modo de adelanto del capítulo 9, voy a usar matplotlib para crear visualizaciones de este array de dos dimensiones:

```
In [174]: import matplotlib.pyplot as plt
```

```
In [175]: plt.imshow(z, cmap=plt.cm.gray, extent=[-5, 5, -5,  
5])
```

```
Out[175]: <matplotlib.image.AxesImage at 0x7f624ae73b20>
```

```
In [176]: plt.colorbar()
```

```
Out[176]: <matplotlib.colorbar.Colorbar at 0x7f6253e43ee0>
```



```
In [177]: plt.title("Image plot of  $\sqrt{x^2 + y^2}$  for a  
grid of values")  
Out[177]: Text(0.5, 1.0, 'Image plot of  $\sqrt{x^2 + y^2}$   
for a grid of values')  
)
```

En la figura 4.3 he empleado la función de matplotlib imshow para crear una representación visual de un array de valores de función de dos dimensiones.

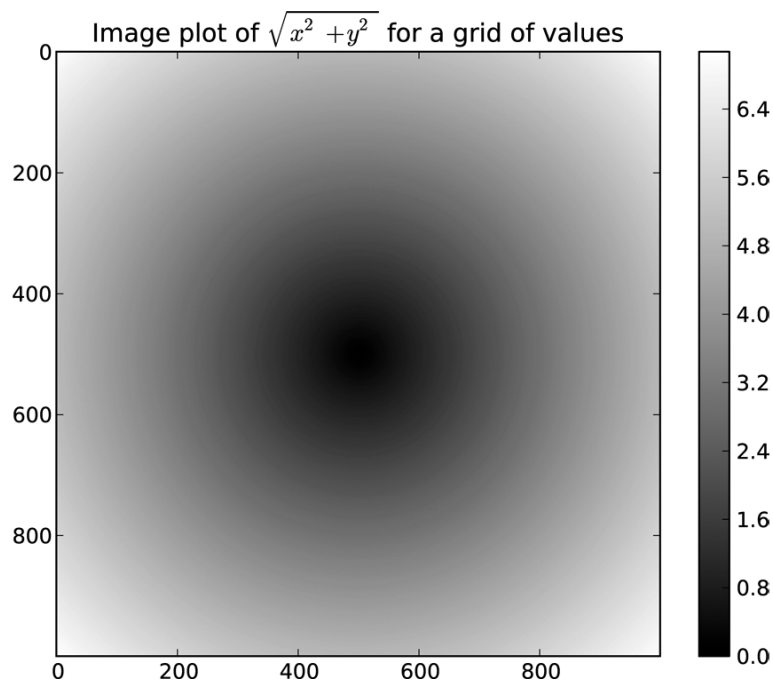


Figura 4.3. Representación de una función evaluada en una cuadrícula.

Si estamos trabajando en IPython, se pueden cerrar todas las ventanas de gráfico abiertas ejecutando `plt.close("all")`:

```
In [179]: plt.close("all")
```



El término vectorización se emplea para describir otros conceptos de ciencia computacional, pero en este libro lo usaremos para describir operaciones realizadas sobre arrays enteros de datos, en lugar de ir valor a valor con un bucle loop de Python.

Expresar lógica condicional como operaciones de arrays

La función `numpy.where` es una versión vectorizada de la expresión ternaria `x if condition else y`. Supongamos que tenemos un array booleano y dos arrays de valores:

```
In [180]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
```

```
In [181]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
```

```
In [182]: cond = np.array([True, False, True, True, False])
```

Imaginemos que queremos tomar un valor de `xarr` siempre que el valor correspondiente de `cond` sea `True`, y en otro caso tomar el valor de `yarr`. Una comprensión de lista que haga esto podría ser algo así:

```
In [183]: result = [(x if c else y)
.....:               for x, y, c in zip(xarr, yarr, cond)]
```

```
In [184]: result
```

```
Out[184]: [1.1, 2.2, 1.3, 1.4, 2.5]
```

Esto tiene varios problemas. Primero, no será muy rápido para arrays grandes (porque todo el trabajo se está realizando en el código interpretado de Python). Segundo, no funcionará con arrays multidimensionales. Con `numpy.where` se puede hacer esto con una sencilla llamada a una función:

```
In [185]: result = np.where(cond, xarr, yarr)
```

```
In [186]: result
```

```
Out[186]: array([1.1, 2.2, 1.3, 1.4, 2.5])
```

Los argumentos segundo y tercero de `numpy.where` no tienen por qué ser arrays; uno de ellos o los dos pueden ser escalares. Un uso habitual de `where` en análisis de datos es producir un nuevo array de valores basado en otro array. Supongamos que tenemos una matriz de datos generados aleatoriamente y queremos reemplazar todos los valores positivos por un 2 y todos los valores negativos por un `-2`. Esto se puede hacer con `numpy.where`:

```
In [187]: arr = rng.standard_normal((4, 4))
```

```
In [188]: arr
```

```
Out[188]:
```

```
array([[ 2.6182,  0.7774,  0.8286, -0.959 ],
       [-1.2094, -1.4123,  0.5415,  0.7519],
       [-0.6588, -1.2287,  0.2576,  0.3129],
       [-0.1308,  1.27  , -0.093  , -0.0662]])
```

```
In [189]: arr > 0
```

```
Out[189]:
```

```
array([[ True,  True,  True, False],
       [False, False,  True,  True],
       [False, False,  True,  True],
       [False,  True, False, False]])
```

```
In [190]: np.where(arr > 0, 2, -2)
```

```
Out[190]:
```

```
array([[ 2,  2,  2, -2],
       [-2, -2,  2,  2],
       [-2, -2,  2,  2],
       [-2,  2, -2, -2]])
```

Se pueden combinar escalares y arrays cuando se utiliza `numpy.where`. Por ejemplo, podemos reemplazar todos los valores positivos de `arr` por la constante 2, de este modo:

```
In [191]: np.where(arr > 0, 2, arr) # fija en 2 solo los valores positivos
```

```
Out[191]:
```

```
array([[ 2. ,  2. ,  2. , -0.959 ],
       [-1.2094, -1.4123,  2. ,  2. ],
       [-0.6588, -1.2287,  2. ,  2. ],
       [-0.1308,  2. , -0.093  , -0.0662]])
```

Métodos matemáticos y estadísticos

Es posible acceder a un conjunto de funciones matemáticas, que calculan estadísticas sobre un array completo o sobre los datos de un eje, como métodos de la clase array. Se pueden emplear agregaciones (a veces llamadas reducciones) como sum, mean y std (desviación estándar) o bien llamando al método de instancia de array o utilizando la función NumPy de máximo nivel. Cuando se emplea la función NumPy, como `numpy.sum`, hay que pasar el array que se desea agregar como primer argumento.

En este fragmento de código se generan datos aleatorios normalmente distribuidos y se calculan ciertas estadísticas agregadas:

```
In [192]: arr = rng.standard_normal((5, 4))
```

```
In [193]: arr
```

```
Out[193]:
```

```
array([[ -1.1082,  0.136 ,  1.3471,  0.0611],  
       [  0.0709,  0.4337,  0.2775,  0.5303],  
       [  0.5367,  0.6184, -0.795 ,  0.3 ],  
       [-1.6027,  0.2668, -1.2616, -0.0713],  
       [  0.474 , -0.4149,  0.0977, -1.6404]])
```

```
In [194]: arr.mean()
```

```
Out[194]: -0.08719744457434529
```

```
In [195]: np.mean(arr)
```

```
Out[195]: -0.08719744457434529
```

```
In [196]: arr.sum()
```

```
Out[196]: -1.743948891486906
```

Funciones como mean y sum toman un argumento axis opcional, que calcula la estadística sobre el eje dado, dando como resultado un array con una dimensión menos:

```
In [197]: arr.mean(axis=1)
```

```
Out[197]: array([ 0.109 ,  0.3281,  0.165 , -0.6672, -0.3709])
```

```
In [198]: arr.sum(axis=0)
```

```
Out[198]: array([-1.6292,  1.0399, -0.3344, -0.8203])
```

En este caso `arr.mean(axis=1)` significa «calcula la media a lo largo de las columnas», mientras que `arr.sum(axis=0)` significa «calcula la suma a lo largo de las filas».

Otros métodos, como `cumsum` y `cumprod`, no agregan; lo que hacen es producir un array de los resultados intermedios:

```
In [199]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
In [200]: arr.cumsum()
```

```
Out[200]: array([ 0, 1, 3, 6, 10, 15, 21, 28])
```

En arrays multidimensionales, funciones de acumulación, como `cumsum`, devuelven un array del mismo tamaño con las sumas parciales calculadas a lo largo del eje indicado, de acuerdo con cada corte dimensional inferior:

```
In [201]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
```

```
In [202]: arr
```

```
Out[202]:
```

```
array([[0, 1, 2],  
       [3, 4, 5],  
       [6, 7, 8]])
```

La expresión `arr.cumsum(axis=0)` calcula la suma acumulada a lo largo de las filas, mientras que `arr.cumsum(axis=1)` calcula las sumas a lo largo de las columnas:

```
In [203]: arr.cumsum(axis=0)
```

```
Out[203]:
```

```
array([[ 0,  1,  2],  
       [ 3,  5,  7],  
       [ 9, 12, 15]])
```

```
In [204]: arr.cumsum(axis=1)
```

```
Out[204]:
```

```
array([[ 0,  1,  3],  
       [ 3,  7, 12],  
       [ 6, 13, 21]])
```

Consulte en la tabla 4.6 un listado completo. Veremos muchos ejemplos de estos métodos en acción en los capítulos posteriores.

Tabla 4.6. Métodos estadísticos de array básicos.

Método	Descripción
sum	Suma de todos los elementos del array o a lo largo de un eje; los arrays de longitud cero tienen suma 0.
mean	Media aritmética; no es válida (devuelve NaN) en arrays de longitud cero.
std, var	Desviación estándar y varianza, respectivamente.
min, max	Mínimo y máximo.
argmin, argmax	Índices de los elementos mínimo y máximo, respectivamente.
cumsum	Suma acumulada de los elementos a partir de 0.
cumprod	Producto acumulado de los elementos a partir de 1.

Métodos para arrays booleanos

En los métodos anteriores, los valores booleanos son forzados al valor 1 (True) y 0 (False). De este modo, sum se utiliza a menudo como una forma de contar valores True en un array booleano:

```
In [205]: arr = rng.standard_normal(100)
```

```
In [206]: (arr > 0).sum() # Número de valores positivos  
Out[206]: 48
```

```
In [207]: (arr <= 0).sum() # Número de valores no positivos  
Out[207]: 52
```

Los paréntesis de la expresión `(arr > 0).sum()` son necesarios para poder llamar a `sum()` en el resultado temporal de `arr > 0`.

Dos métodos adicionales, `any` y `all`, son útiles especialmente para arrays booleanos. `any` verifica si uno o varios valores de un array es True, mientras

que `all` comprueba que todos los valores son `True`:

```
In [208]: bools = np.array([False, False, True, False])
```

```
In [209]: bools.any()
```

```
Out[209]: True
```

```
In [210]: bools.all()
```

```
Out[210]: False
```

Estos métodos funcionan también con arrays no booleanos, donde los elementos que no son cero son tratados como `True`.

Ordenación

Al igual que el tipo de lista interno de Python, los arrays NumPy pueden ordenarse en el momento con el método `sort`:

```
In [211]: arr = rng.standard_normal(6)
```

```
In [212]: arr
```

```
Out[212]: array([ 0.0773, -0.6839, -0.7208, 1.1206, -0.0548,  
-0.0824])
```

```
In [213]: arr.sort()
```

```
In [214]: arr
```

```
Out[214]: array([-0.7208, -0.6839, -0.0824, -0.0548, 0.0773,  
1.1206])
```

Se puede ordenar cada sección unidimensional de valores de un array multidimensional en el momento a lo largo de un eje pasando el número de eje a ordenar. En estos datos de ejemplo:

```
In [215]: arr = rng.standard_normal((5, 3))
```

```
In [216]: arr
```

```
Out[216]:
```

```
array([[ 0.936 , 1.2385, 1.2728],  
[ 0.4059, -0.0503, 0.2893],  
[ 0.1793, 1.3975, 0.292 ],
```

```
[ 0.6384, -0.0279, 1.3711],  
[-2.0528, 0.3805, 0.7554]])
```

`arr.sort(axis=0)` ordena los valores dentro de cada columna, mientras que `arr.sort(axis=1)` los ordena a lo largo de cada fila:

```
In [217]: arr.sort(axis=0)
```

```
In [218]: arr
```

```
Out[218]:
```

```
array([[ -2.0528, -0.0503,  0.2893],  
[  0.1793, -0.0279,  0.292 ],  
[  0.4059,  0.3805,  0.7554],  
[  0.6384,  1.2385,  1.2728],  
[  0.936 ,  1.3975,  1.3711]])
```

```
In [219]: arr.sort(axis=1)
```

```
In [220]: arr
```

```
Out[220]:
```

```
array([[ -2.0528, -0.0503,  0.2893],  
[ -0.0279,  0.1793,  0.292 ],  
[  0.3805,  0.4059,  0.7554],  
[  0.6384,  1.2385,  1.2728],  
[  0.936 ,  1.3711,  1.3975]])
```

El método `numpy.sort` de máximo nivel devuelve una copia ordenada de un array (igual que la función `sorted` interna de Python), en vez de modificar el array en el momento. Por ejemplo:

```
In [221]: arr2 = np.array([5, -10, 7, 1, 0, -3])
```

```
In [222]: sorted_arr2 = np.sort(arr2)
```

```
In [223]: sorted_arr2
```

```
Out[223]: array([-10, -3, 0, 1, 5, 7])
```

Para más detalles sobre el uso de métodos de ordenación de NumPy y sobre técnicas más avanzadas, como las ordenaciones indirectas, consulte el

apéndice A. También pueden encontrarse en pandas otros tipos de manipulaciones de datos relacionados con la ordenación (por ejemplo, ordenar una tabla de datos por una o varias columnas).

Unique y otra lógica de conjuntos

NumPy tiene varias operaciones de conjuntos básicas para ndarrays unidimensionales. Una que se utiliza mucho es `numpy.unique`, que devuelve los valores únicos de un array ordenados:

```
In [224]: names = np.array(["Bob", "Will", "Joe", "Bob",  
"Will", "Joe", "Joe"])
```

```
In [225]: np.unique(names)  
Out[225]: array(['Bob', 'Joe', 'Will'], dtype='<U4')
```

```
In [226]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
```

```
In [227]: np.unique(ints)  
Out[227]: array([1, 2, 3, 4])
```

Podemos comparar `numpy.unique` con la alternativa pura de Python:

```
In [228]: sorted(set(names))  
Out[228]: ['Bob', 'Joe', 'Will']
```

En muchos casos, la versión de NumPy es más rápida y devuelve un array NumPy en lugar de una lista Python.

Otra función, `numpy.in1d`, prueba la membresía de los valores de un array en otro, devolviendo un array booleano:

```
In [229]: values = np.array([6, 0, 0, 3, 2, 5, 6])
```

```
In [230]: np.in1d(values, [2, 3, 6])  
Out[230]: array([ True, False, False,  True,  True, False,  
 True])
```

Véase en la tabla 4.7 un listado de operaciones de conjuntos de arrays en NumPy.

Tabla 4.7. Operaciones de conjuntos de array.

Método	Descripción
<code>unique(x)</code>	Calcula los elementos únicos ordenados de <code>x</code> .
<code>intersect1d(x, y)</code>	Calcula los elementos comunes ordenados de <code>x</code> e <code>y</code> .
<code>union1d(x, y)</code>	Calcula la unión ordenada de elementos.
<code>in1d(x, y)</code>	Calcula un array booleano que indica si cada elemento de <code>x</code> está contenido en <code>y</code> .
<code>setdiff1d(x, y)</code>	Establece la diferencia; los elementos de <code>x</code> que no están en <code>y</code> .
<code>setxor1d(x, y)</code>	Establece las diferencias simétricas; elementos que están en alguno de los arrays, pero no en los dos.

4.5 Entrada y salida de archivos con arrays

NumPy es capaz de guardar y cargar datos en disco en varios formatos de texto o binarios. En esta sección hablaré solo del formato binario interno de NumPy, ya que la mayor parte de los usuarios preferirán pandas y otras herramientas para cargar texto o datos tabulares (consulte el capítulo 6 si desea más información).

`numpy.save` y `numpy.load` son las dos funciones principales para guardar y cargar datos en disco de forma eficaz. Los arrays se guardan por omisión en un formato binario sin procesar y no comprimido con la extensión `.npy`:

```
In [231]: arr = np.arange(10)
```

```
In [232]: np.save("some_array", arr)
```

Si la ruta del archivo no termina ya en `.npy`, la extensión será añadida. El array en disco se puede cargar entonces con `numpy.load`:

```
In [233]: np.load("some_array.npy")
Out[233]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Se pueden guardar varios arrays en un archivo no comprimido usando `numpy.savez` y pasando los arrays como argumentos de palabra clave:

```
In [234]: np.savez("array_archive.npz", a=arr, b=arr)
```

Cuando se carga un archivo `.npz`, se obtiene de vuelta un objeto de estilo diccionario que carga los arrays individuales de una forma un tanto indolente:

```
In [235]: arch = np.load("array_archive.npz")
```

```
In [236]: arch["b"]
```

```
Out[236]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Si los datos se comprimen bien, puede que convenga más usar `numpy.savez_compressed`:

```
In [237]: np.savez_compressed("arrays_compressed.npz",  
a=arr, b=arr)
```

4.6 Álgebra lineal

Las operaciones de álgebra lineal, como la multiplicación de matrices, descomposiciones, determinantes y otros cálculos matemáticos de matrices cuadradas, son una parte importante de muchas librerías de arrays. Multiplicar dos arrays bidimensionales con `*` es un producto elemento a elemento, mientras que las multiplicaciones de matrices requieren el uso de una función. Así, para la multiplicación de matrices existe una función `dot`, un método `array` y una función en el espacio de nombres `numpy`:

```
In [241]: x = np.array([[1., 2., 3.], [4., 5., 6.]])
```

```
In [242]: y = np.array([[6., 23.], [-1, 7], [8, 9]])
```

```
In [243]: x
```

```
Out[243]:
```

```
array([[1., 2., 3.],  
       [4., 5., 6.]])
```

```
In [244]: y
Out[244]:
array([[ 6., 23.],
       [-1.,  7.],
       [ 8.,  9.]])
```

```
In [245]: x.dot(y)
Out[245]:
array([[ 28., 64.],
       [ 67., 181.]])
```

`x.dot(y)` es equivalente a `np.dot(x, y)`:

```
In [246]: np.dot(x, y)
Out[246]:
array([[ 28., 64.],
       [ 67., 181.]])
```

Un producto de matrices entre un array bidimensional y un array unidimensional del tamaño adecuado da como resultado otro array unidimensional:

```
In [247]: x @ np.ones(3)
Out[247]: array([ 6., 15.])
```

`numpy.linalg` tiene un conjunto estándar de descomposiciones matriciales, y utilidades como la inversa y el determinante:

```
In [248]: from numpy.linalg import inv, qr
```

```
In [249]: X = rng.standard_normal((5, 5))
```

```
In [250]: mat = X.T @ X
```

```
In [251]: inv(mat)
Out[251]:
array([[ 3.4993,  2.8444,  3.5956, -16.5538,  4.4733],
       [ 2.8444,  2.5667,  2.9002, -13.5774,  3.7678],
       [ 3.5956,  2.9002,  4.4823, -18.3453,  4.7066],
       [-16.5538, -13.5774, -18.3453, 84.0102, -22.0484],
       [ 4.4733,  3.7678,  4.7066, -22.0484, 10.5555]])
```

```
[ 4.4733, 3.7678, 4.7066, -22.0484, 6.0525]])
```

```
In [252]: mat @ inv(mat)
Out[252]:
array([[ 1.,  0., -0.,  0., -0.],
       [ 0.,  1.,  0.,  0., -0.],
       [ 0., -0.,  1., -0., -0.],
       [ 0., -0.,  0.,  1., -0.],
       [ 0., -0.,  0., -0.,  1.]])
```

La expresión `x.T.dot(x)` calcula el producto punto de `x` con su transposición `x.T`.

Consulte en la tabla 4.8 una lista de algunas de las funciones de álgebra lineal más utilizadas.

Tabla 4.8. Funciones `numpy.linalg` más utilizadas.

Función	Descripción
<code>diag</code>	Devuelve los elementos de la diagonal (o de fuera de la diagonal) de una matriz cuadrada como un array de una dimensión, o convierte un array unidimensional en una matriz cuadrada con ceros fuera de la diagonal.
<code>dot</code>	Multiplicación de matrices.
<code>trace</code>	Calcula la suma de los elementos de la diagonal.
<code>det</code>	Calcula el determinante de la matriz.
<code>eig</code>	Calcula los valores propios y vectores propios de una matriz cuadrada.
<code>inv</code>	Calcula la inversa de una matriz cuadrada.
<code>pinv</code>	Calcula la pseudoinversa de Moore-Penrose de una matriz.
<code>qr</code>	Calcula la descomposición o factorización QR.
<code>svd</code>	Calcula la descomposición en valores singulares o DVS.
<code>solve</code>	Resuelve el sistema lineal $Ax = b$ para x , donde A es una matriz cuadrada.
<code>lstsq</code>	Calcula la solución de mínimos cuadrados a $Ax = b$.

4.7 Ejemplo: caminos aleatorios

La simulación de caminos aleatorios (https://es.wikipedia.org/wiki/Camino_aleatorio) ofrece una aplicación ilustrativa del uso de operaciones con arrays. Consideremos en primera instancia un sencillo camino aleatorio que comienza en 0 y en el cual se producen incrementos de 1 y -1 con la misma probabilidad.

Aquí tenemos una forma pura de Python de implementar un solo camino aleatorio con 1000 pasos utilizando el módulo `random` integrado:

```
#!/ blockstart
import random
position = 0
walk = [position]
nsteps = 1000
for _ in range(nsteps):
    step = 1 if random.randint(0, 1) else -1
    position += step
    walk.append(position)
#!/ blockend
```

Véase en la figura 4.4 un gráfico de ejemplo de los primeros 100 valores de uno de estos caminos aleatorios:

```
In [255]: plt.plot(walk[:100])
```

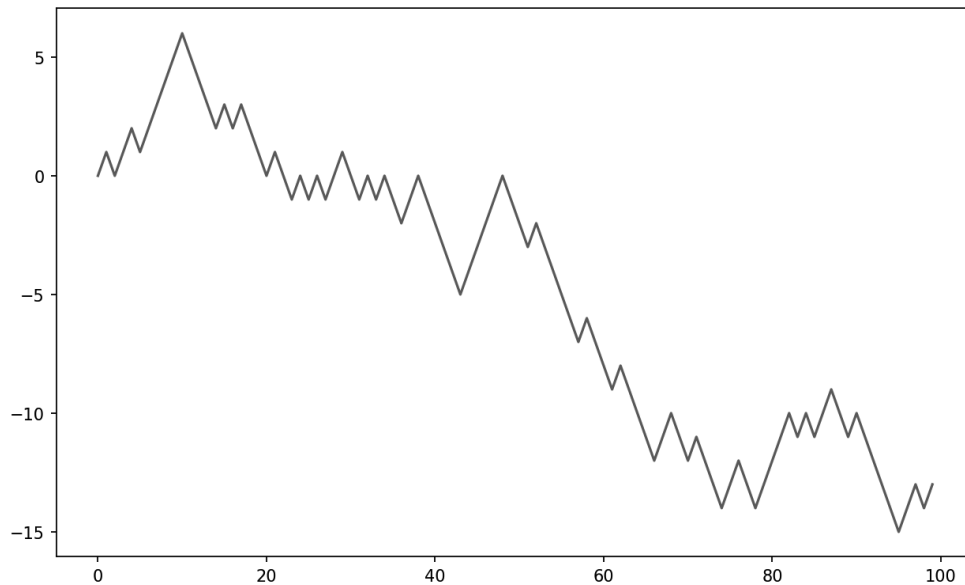


Figura 4.4. Un sencillo camino aleatorio.

Quizá haya resultado fácil observar que `walk` es la suma acumulada de los pasos aleatorios y que se podría evaluar como expresión array. Por esa razón usamos el módulo `numpy.random` para lanzar 1000 veces una moneda al mismo tiempo, establecer los lanzamientos en 1 y -1, y calcular la suma acumulada:

```
In [256]: nsteps = 1000
```

```
In [257]: rng = np.random.default_rng(seed=12345) #  
# generador aleatorio nuevo
```

```
In [258]: draws = rng.integers(0, 2, size=nsteps)
```

```
In [259]: steps = np.where(draws == 0, 1, -1)
```

```
In [260]: walk = steps.cumsum()
```

A partir de aquí podemos empezar a extraer estadísticas como el valor mínimo y máximo a lo largo de la trayectoria del camino:

```
In [261]: walk.min()  
Out[261]: -8
```

```
In [262]: walk.max()  
Out[262]: 50
```

Una estadística más compleja es el tiempo de primer cruce, el paso en el que el camino aleatorio alcanza un determinado valor. Aquí queremos saber cuánto tardará el camino aleatorio en llegar al menos 10 pasos más allá del 0 original en cualquier dirección. `np.abs(walk) >= 10` nos da un array booleano indicando el punto en el que el camino ha llegado a 0 o lo ha superado, pero queremos el índice de los primeros 10 o -10. Resulta que podemos calcular esto utilizando `argmax`, que devuelve el primer índice del valor máximo del array booleano (True es el valor máximo):

```
In [263]: (np.abs(walk) >= 10).argmax()  
Out[263]: 155
```

Hay que tener en cuenta que usar aquí `argmax` no es siempre eficaz, porque en todas las ocasiones realiza una exploración completa del array. En este caso especial, una vez que se observa un True, sabemos que es el valor máximo.

Simulando muchos caminos aleatorios al mismo tiempo

Si el objetivo es simular muchos caminos aleatorios, digamos miles de ellos, es posible generarlos todos con modificaciones menores sobre el código anterior. Si se pasó una tupla de dos, las funciones `numpy.random` generarán un array bidimensional de lanzamientos, y podremos obtener la suma acumulada para cada fila para calcular los cinco caminos aleatorios de una sola vez:

```
In [264]: nwalks = 5000  
  
In [265]: nsteps = 1000  
  
In [266]: draws = rng.integers(0, 2, size=(nwalks, nsteps))  
# 0 or 1  
  
In [267]: steps = np.where(draws > 0, 1, -1)  
  
In [268]: walks = steps.cumsum(axis=1)  
  
In [269]: walks
```



```
Out[269]:
array([[ 1,  2,  3, ..., 22, 23, 22],
       [ 1,  0, -1, ..., -50, -49, -48],
       [ 1,  2,  3, ..., 50, 49, 48],
       ...,
       [-1, -2, -1, ..., -10, -9, -10],
       [-1, -2, -3, ...,  8,  9,  8],
       [-1,  0,  1, ..., -4, -3, -2]])
```

Ahora podemos calcular los valores máximo y mínimo obtenidos en todos los caminos:

```
In [270]: walks.max()
Out[270]: 114
```

```
In [271]: walks.min()
Out[271]: -120
```

Fuera de ellos, calculemos el tiempo de cruce mínimo en 30 o -30. Esto es un poco difícil, porque no todos los 5000 llegan a 30. Lo podemos comprobar utilizando el método any:

```
In [272]: hits30 = (np.abs(walks) >= 30).any(axis=1)
```

```
In [273]: hits30
Out[273]: array([False,  True,  True, ...,  True, False,  True])
```

```
In [274]: hits30.sum() # Número que alcanza 30 o -30
Out[274]: 3395
```

Podemos emplear este array booleano para elegir las filas de caminos que realmente cruzan el nivel absoluto 30, y podemos llamar a argmax a lo largo del eje 1 para obtener los tiempos de cruce:

```
In [275]: crossing_times = (np.abs(walks[hits30]) >=
30).argmax(axis=1)
```

```
In [276]: crossing_times
Out[276]: array([201, 491, 283, ..., 219, 259, 541])
```

Por último, calculamos el tiempo de cruce medio mínimo:

```
In [277]: crossing_times.mean()  
Out[277]: 500.5699558173785
```

Experimente como desee con otras distribuciones para los pasos que no sean lanzamientos de monedas de igual tamaño. Únicamente será necesario utilizar un método de generación aleatoria diferente, como `standard_normal`, para generar pasos normalmente distribuidos con una cierta media y desviación estándar:

```
In [278]: draws = 0.25 * rng.standard_normal((nwalks,  
nsteps))
```



Recuerde que este enfoque vectorizado requiere la creación de un array con `nwalks * nsteps` elementos, que pueden usar una cantidad de memoria grande para simulaciones grandes. Si la memoria es limitada, será necesario emplear un enfoque distinto.

4.8 Conclusión

Aunque buena parte del resto del libro se centrará en la creación de habilidades de manipulación de datos con pandas, seguiremos trabajando en un estilo similar basado en arrays. En el apéndice A, profundizaremos más en las funciones NumPy para que pueda desarrollar aún más sus habilidades de cálculo de arrays.

Empezar a trabajar con pandas

Manejaremos mucho la herramienta pandas durante buena parte del resto del libro. Contiene estructuras de datos y herramientas de manipulación de datos diseñadas para que la limpieza y el análisis de los datos sean rápidos y cómodos en Python. Con bastante frecuencia se emplea en colaboración con herramientas de cálculo numérico, como NumPy y SciPy; librerías analíticas, como statsmodels y scikit-learn, y librerías de visualización de datos, como matplotlib. En algunas de sus partes más importantes, la librería pandas sigue el estilo idiomático de la computación basada en arrays de NumPy, especialmente en lo referente a las funciones basadas en arrays y a su preferencia por el proceso de datos sin utilizar bucles for.

Aunque adopte buena parte de las expresiones de codificación de NumPy, la gran diferencia entre ellos es que pandas está diseñada para trabajar con datos tabulares o heterogéneos. NumPy, por el contrario, es más adecuada para trabajar con datos de arrays numéricos de tipos homogéneos.

Desde que se convirtió en 2010 en un proyecto de código abierto, pandas se ha transformado en una librería de gran envergadura, que se puede aplicar a un amplio conjunto de situaciones de uso reales. Su comunidad de desarrolladores ha crecido hasta estar formada por más de 2500 colaboradores, quienes han ayudado a crear el proyecto a medida que lo utilizaban para resolver sus problemas de datos cotidianos. La gran actividad mostrada por las comunidades de desarrolladores y usuarios de pandas las ha erigido en una parte fundamental de su éxito.



Mucha gente no sabe que llevo desde 2013 sin estar implicado activamente en el desarrollo continuado de pandas; desde entonces ha sido un proyecto totalmente gestionado por su comunidad. No deje de agradecer su gran trabajo a sus desarrolladores y colaboradores.

Durante el resto del libro, voy a emplear los siguientes convenios de importación para NumPy y pandas:

```
In [1]: import numpy as np
```

```
In [2]: import pandas as pd
```

Así, siempre que vea `pd.` en el código, sabrá que se estará refiriendo a pandas. Quizá también le resulte sencillo importar objetos `Series` y `DataFrame` en el espacio de nombres local, dado que se utilizan con tanta frecuencia:

```
In [3]: from pandas import Series, DataFrame
```

5.1 Introducción a las estructuras de datos de pandas

Para empezar a trabajar con pandas, es conveniente sentirse cómodo con sus dos estructuras de datos principales: `Series` y `DataFrame`. Aunque no son la solución universal a todos los problemas, ofrecen una sólida base para una amplia variedad de tareas de datos.

Series

Una serie es un objeto unidimensional de estilo array, que contiene una secuencia de valores (de tipos parecidos a los de NumPy) del mismo tipo y un array asociado de etiquetas de datos, que corresponde a su índice. El objeto `Series` más sencillo está formado únicamente por un array de datos:

```
In [14]: obj = pd.Series([4, 7, -5, 3])
```

```
In [15]: obj
```

```
Out[15]:
```

0	4
1	7
2	-5
3	3
dtype: int64	

La representación como cadena de texto de una serie, visualizada interactivamente, muestra el índice a la izquierda y los valores a la derecha. Como no especificamos un índice para los datos, se crea uno predeterminado, formado por los enteros 0 a $N - 1$ (donde N es la longitud de los datos). Se puede obtener la representación en array y el objeto índice de la serie mediante sus atributos `array` e `index`, respectivamente:

```
In [16]: obj.array
Out[16]:
<PandasArray>
[4, 7, -5, 3]
Length: 4, dtype: int64

In [17]: obj.index
Out[17]: RangeIndex(start=0, stop=4, step=1)
```

El resultado del atributo `.array` es un `PandasArray`, que normalmente encierra un array `NumPy`, pero puede contener además tipos de array de extensión especial, de los que hablaremos con más detalles en la sección 7.3 «Tipos de datos de extensión».

Con frecuencia nos interesará crear una serie con un índice, que identifique cada punto de datos con una etiqueta:

```
In [18]: obj2 = pd.Series([4, 7, -5, 3], index=["d", "b",
"a", "c"])

In [19]: obj2
Out[19]:
d                                4
b                                7
a                               -5
c                                 3
dtype: int64

In [20]: obj2.index
Out[20]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

Comparando con los arrays `NumPy`, se pueden usar etiquetas en el índice al seleccionar valores sencillos o un conjunto de valores:

```
In [21]: obj2["a"]
```

```
Out[21]: -5
```

```
In [22]: obj2["d"] = 6
```

```
In [23]: obj2[["c", "a", "d"]]
```

```
Out[23]:
```

```
c 3
```

```
a -5
```

```
d 6
```

```
dtype: int64
```

Aquí, ["c", "a", "d"] se interpreta como una lista de índices, aunque contenga cadenas de texto en lugar de enteros.

Utilizar funciones NumPy u operaciones de estilo NumPy, como el filtrado con un array booleano, la multiplicación de escalares o la aplicación de funciones matemáticas, permitirá conservar el vínculo índice-valor:

```
In [24]: obj2[obj2 > 0]
```

```
Out[24]:
```

```
d 6
```

```
b 7
```

```
c 3
```

```
dtype: int64
```

```
In [25]: obj2 * 2
```

```
Out[25]:
```

```
d 12
```

```
b 14
```

```
a -10
```

```
c 6
```

```
dtype: int64
```

```
In [26]: import numpy as np
```

```
In [27]: np.exp(obj2)
```

```
Out[27]:
```

```
d 403.428793
```

```
b 1096.633158
```

```
a                                0.006738
c                                20.085537
```

```
dtype: float64
```

Otra forma de pensar en una serie es como si fuera un diccionario ordenado de longitud fija, dado que es una asignación de valores de índice a valores de datos. Se puede emplear en muchos contextos en los que se podría usar un diccionario:

```
In [28]: "b" in obj2
Out[28]: True
```

```
In [29]: "e" in obj2
Out[29]: False
```

Si tenemos datos contenidos en un diccionario Python, podemos crear una serie a partir de él pasando el diccionario:

```
In [30]: sdata = {"Ohio": 35000, "Texas": 71000, "Oregon": 16000, "Utah": 5000}
```

```
In [31]: obj3 = pd.Series(sdata)
In [32]: obj3
Out[32]:
```

```
Ohio                35000
Texas               71000
Oregon              16000
Utah                 5000
dtype: int64
```

Una serie se puede convertir de nuevo en un diccionario con su método `to_dict`:

```
In [33]: obj3.to_dict()
Out[33]: {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
```

Cuando solo se pasa un diccionario, el índice de la serie resultante respetará el orden de las claves, de acuerdo con el método `keys` del

diccionario, que depende del orden de inserción de las claves. Esto se puede anular pasando un índice con las claves de diccionario en el orden en el cual se desea que aparezcan en la serie resultante:

```
In [34]: states = ["California", "Ohio", "Oregon", "Texas"]

In [35]: obj4 = pd.Series(sdata, index=states)

In [36]: obj4
Out[36]:
California          NaN
Ohio               35000.0
Oregon             16000.0
Texas              71000.0
dtype: float64
```

Aquí, los tres valores hallados en sdata se colocaron en las ubicaciones adecuadas, pero como no se encontró ningún valor para “California”, aparece como NaN (Not a Number), expresión que en pandas indica valores ausentes o faltantes. Como “Utah” no estaba incluido en states, se excluyó del objeto resultante.

Utilizaré los términos «ausentes», «faltantes» o «nulos» de forma intercambiable para referirme a datos que no están o que faltan. Para detectar estos datos se deben emplear las funciones `isna` y `notna` de pandas:

```
In [37]: pd.isna(obj4)
Out[37]:
California          True
Ohio               False
Oregon             False
Texas              False
dtype: bool
```

```
In [38]: pd.notna(obj4)
Out[38]:
California          False
Ohio               True
Oregon              True
```



```
Texas                                         True
dtype: bool
```

El objeto Series también incluye estas funciones como métodos de instancia:

```
In [39]: obj4.isna()
Out[39]:
California                                     True
Ohio                                           False
Oregon                                         False
Texas                                           False
dtype: bool
```

Hablaré con más detalle del trabajo con datos ausentes en el capítulo 7. Una característica útil del objeto Series para muchas aplicaciones es que en las operaciones aritméticas se alinea automáticamente por etiqueta de índice:

```
In [40]: obj3
Out[40]:
Ohio                                     35000
Texas                                    71000
Oregon                                  16000
Utah                                     5000
dtype: int64
```

```
In [41]: obj4
Out[41]:
California                                     NaN
Ohio                                           35000.0
Oregon                                         16000.0
Texas                                           71000.0
dtype: float64
```

```
In [42]: obj3 + obj4
Out[42]:
```

California	NaN
Ohio	70000.0
Oregon	32000.0
Texas	142000.0
Utah	NaN

dtype: float64

Las funciones de alineación de datos se tratarán con más detalle más adelante. Si tiene experiencia con bases de datos, esto le puede parecer similar a una operación JOIN.

Tanto el objeto Series en sí como su índice tienen un atributo name, que se integra con otras áreas de la funcionalidad de pandas:

```
In [43]: obj4.name = "population"
```

```
In [44]: obj4.index.name = "state"
```

```
In [45]: obj4
```

```
Out[45]:
```

```
state
```

```
California
```

```
NaN
```

```
Ohio
```

```
35000.0
```

```
Oregon
```

```
16000.0
```

```
Texas
```

```
71000.0
```

```
Name: population, dtype: float64
```

El índice de una serie se puede modificar en el momento mediante asignación:

```
In [46]: obj
```

```
Out[46]:
```

```
0
```

```
4
```

```
1
```

```
7
```

```
2
```

```
-5
```

```
3
```

```
3
```

```
dtype: int64
```

```
In [47]: obj.index = ["Bob", "Steve", "Jeff", "Ryan"]
```

```
In [48]: obj
Out[48]:
Bob                                     4
Steve                                  7
Jeff                                  -5
Ryan                                   3
dtype: int64
```

DataFrame

Un dataframe representa una tabla rectangular de datos, y contiene una colección de columnas ordenada y con nombre, cada una de las cuales puede tener un tipo de valor distinto (numérico, cadena de texto, booleano, etc.). El objeto DataFrame tiene un índice de fila y otro de columna; se podría considerar como un diccionario de objetos Series que comparten todos el mismo índice.



Aunque un dataframe tiene físicamente dos dimensiones, se puede utilizar para representar datos de más dimensiones en un formato tabular, empleando la indexación jerárquica, un tema del que hablaremos en el capítulo 8, y un ingrediente de algunas de las características de manejo de datos más avanzadas de pandas.

Hay muchas formas de construir un dataframe, aunque una de las más habituales es a partir de un diccionario de listas o arrays NumPy de la misma longitud:

```
data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada",
                 "Nevada", "Nevada"],
        "year": [2000, 2001, 2002, 2001, 2002, 2003],
        "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

El objeto DataFrame resultante tendrá su índice asignado de manera automática, como con Series, y las columnas se colocarán de acuerdo con el orden de las claves en data (que depende de su orden de inserción en el diccionario):

```
In [50]: frame
```

Out[50]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2



Si está utilizando Jupyter notebook, los objetos DataFrame de pandas se mostrarán como una tabla HTML apta para navegadores. Véase un ejemplo en la figura 5.1.

```
In [19]: frame
```

Out[19]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

Figura 5.1. Aspecto de los objetos DataFrame de pandas en Jupyter.

Para grandes dataframes, el método `head` selecciona solo las cinco primeras filas:

```
In [51]: frame.head()
Out[51]:
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7

2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9

De forma similar, `tail` devuelve las cinco últimas:

```
In [52]: frame.tail()
Out[52]:
```

	state	year	pop
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

Si se especifica una secuencia de columnas, las columnas del dataframe se dispondrán en ese orden:

```
In [53]: pd.DataFrame(data, columns=["year", "state",
"pop"])
Out[53]:
```

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9
5	2003	Nevada	3.2

Si se pasa una columna no contenida en el diccionario, aparecerá con valores faltantes en el resultado:

```
In [54]: frame2 = pd.DataFrame(data, columns=["year",
"state", "pop", "debt"])
```

```
In [55]: frame2
Out[55]:
```

	year	state	pop	debt
0	2000	Ohio	1.5	NaN
1	2001	Ohio	1.7	NaN
2	2002	Ohio	3.6	NaN
3	2001	Nevada	2.4	NaN
4	2002	Nevada	2.9	NaN
5	2003	Nevada	3.2	NaN

```
In [56]: frame2.columns
Out[56]: Index(['year', 'state', 'pop', 'debt'],
dtype='object')
```

Es posible recuperar una columna de un dataframe como una serie o bien mediante la notación de estilo diccionario o utilizando la notación de atributo de punto:

```
In [57]: frame2["state"]
Out[57]:
```


0	Ohio
1	Ohio
2	Ohio
3	Nevada
4	Nevada
5	Nevada

```
Name: state, dtype: object
```

```
In [58]: frame2.year
Out[58]:
```

0	2000
1	2001
2	2002
3	2001
4	2002
5	2003

```
Name: year, dtype: int64
```

 El acceso de estilo atributo (por ejemplo, `frame2.year`) y el completado por tabulación de los nombres de columna en IPython se incluyen por comodidad. `frame2[column]` sirve con cualquier nombre de columna, pero `frame2.column` solo funciona cuando el nombre de la columna es un nombre de variable válido de Python, y no entra en conflicto con ninguno de los nombres de métodos del objeto DataFrame. Por ejemplo, si el nombre de una columna contiene espacios en blanco o símbolos distintos al carácter de subrayado, no se puede acceder a ella con el método de atributo de punto.

Observe que la serie devuelta tiene el mismo índice que el dataframe, y que su atributo `name` ha sido adecuadamente establecido.

Las filas también se pueden recuperar por posición o nombre con los atributos especiales `iloc` y `loc` (más información sobre esto en el apartado «Selección en dataframes con `loc` e `iloc`»):

```
In [59]: frame2.loc[1]
Out[59]:
year                2001
state              Ohio
pop                1.7
debt               NaN
Name: 1, dtype: object
```

```
In [60]: frame2.iloc[2]
Out[60]:
year                2002
state              Ohio
pop                3.6
debt               NaN
Name: 2, dtype: object
```

Las columnas se pueden modificar por asignación. Por ejemplo, a la columna vacía `debt` se le podría asignar un valor escalar o un array de valores:

```
In [61]: frame2["debt"] = 16.5

In [62]: frame2
Out[62]:
```

	year	state	pop	debt
0	2000	Ohio	1.5	16.5
1	2001	Ohio	1.7	16.5
2	2002	Ohio	3.6	16.5
3	2001	Nevada	2.4	16.5
4	2002	Nevada	2.9	16.5
5	2003	Nevada	3.2	16.5

```
In [63]: frame2["debt"] = np.arange(6.)
```

```
In [64]: frame2
```

```
Out[64]:
```

	year	state	pop	debt
0	2000	Ohio	1.5	0.0
1	2001	Ohio	1.7	1.0
2	2002	Ohio	3.6	2.0
3	2001	Nevada	2.4	3.0
4	2002	Nevada	2.9	4.0
5	2003	Nevada	3.2	5.0

Cuando se asignan listas o arrays a una columna, la longitud del valor debe coincidir con la longitud del dataframe. Si se asigna una serie, sus etiquetas se realinearán exactamente con el índice del dataframe, insertando los valores faltantes en cualesquiera valores de índice no presentes:

```
In [65]: val = pd.Series([-1.2, -1.5, -1.7], index=["two",
"four", "five"])
```

```
In [66]: frame2["debt"] = val
```

```
In [67]: frame2
```

```
Out[67]:
```

	year	state	pop	debt
0	2000	Ohio	1.5	NaN
1	2001	Ohio	1.7	NaN
2	2002	Ohio	3.6	NaN
3	2001	Nevada	2.4	NaN

4	2002	Nevada	2.9	NaN
5	2003	Nevada	3.2	NaN

Asignar una columna que no existe creará una nueva columna.

La palabra clave `del` borrará columnas, como con un diccionario. Como ejemplo, primero añado una nueva columna de valores booleanos donde la columna `state` es igual a "Ohio":

```
In [68]: frame2["eastern"] = frame2["state"] == "Ohio"
```

```
In [69]: frame2
```

```
Out[69]:
```

	year	state	pop	debt	eastern
0	2000	Ohio	1.5	NaN	True
1	2001	Ohio	1.7	NaN	True
2	2002	Ohio	3.6	NaN	True
3	2001	Nevada	2.4	NaN	False
4	2002	Nevada	2.9	NaN	False
5	2003	Nevada	3.2	NaN	False



No se pueden crear nuevas columnas con la notación de atributo de punto `frame2.eastern`.

El método `del` se puede emplear después para eliminar esta columna:

```
In [70]: del frame2["eastern"]
```

```
In [71]: frame2.columns
```

```
Out[71]: Index(['year', 'state', 'pop', 'debt'],
              dtype='object')
```



La columna devuelta al indexar un dataframe es una vista de los datos subyacentes, no una copia. Por lo tanto, cualquier modificación que se haga en ese momento sobre la serie se verá

reflejada en el dataframe. La columna se puede copiar de forma explícita con el método `copy` del objeto `Series`.

Otra forma habitual de datos es un diccionario anidado de diccionarios:

```
In [72]: populations = {"Ohio": {2000: 1.5, 2001: 1.7,
.....:                  2002: 3.6},
.....:                  "Nevada": {2001: 2.4, 2002: 2.9}}
```

Si el diccionario anidado se pasa al dataframe, pandas interpretará las claves de diccionario externas como las columnas, y las internas como los índices de fila:

```
In [73]: frame3 = pd.DataFrame(populations)
```

```
In [74]: frame3
```

```
Out[74]:
```

	Ohio	Nevada
2000	1.5	NaN
2001	1.7	2.4
2002	3.6	2.9

Es posible transponer el dataframe (intercambiar filas y columnas) con una sintaxis similar a la de un array NumPy:

```
In [75]: frame3.T
```

```
Out[75]:
```

	2000	2001	2002
Ohio	1.5	1.7	3.6
Nevada	NaN	2.4	2.9



Conviene tener en cuenta que la transposición descarta los tipos de datos de las columnas si estas no tienen todas el mismo tipo de datos, de modo que transponerlas y después cancelar la transposición conseguirá que se pierda la información anterior de tipos. En este caso las columnas se convierten en arrays de objetos Python puros.

Las claves de los diccionarios internos se combinan para formar el índice del resultado. Esto no aplica si se especifica un índice explícito:

```
In [76]: pd.DataFrame(populations, index=[2001, 2002, 2003])
Out[76]:
```

	Ohio	Nevada
2001	1.7	2.4
2002	3.6	2.9
2003	NaN	NaN

Los diccionarios de series se tratan de una forma muy parecida:

```
In [77]: pdata = {"Ohio": frame3["Ohio"][: -1],
.....:          "Nevada": frame3["Nevada"][:2]}
```

```
In [78]: pd.DataFrame(pdata)
Out[78]:
```

	Ohio	Nevada
2000	1.5	NaN
2001	1.7	2.4

Véase en la tabla 5.1 una lista de muchos de los elementos que se le pueden pasar a un constructor DataFrame.

Tabla 5.1. Posibles entradas de datos para el constructor DataFrame.

Tipo	Notas
ndarray de dos dimensiones	Una matriz de datos, que pasa etiquetas de fila y columna opcionales.
Diccionario de arrays, listas o tuplas	Cada secuencia se convierte en una columna en el dataframe; todas las secuencias deben tener la misma longitud.
Array NumPy estructurado/de registro	Tratado igual que el tipo «diccionario de arrays».
Diccionario de series	Cada valor se convierte en una columna; los índices de cada serie se unen entre sí para formar el índice de fila del resultado si no se le pasa un índice explícito.

Tipo	Notas
Diccionario de diccionarios	Cada diccionario interno se convierte en una columna; las claves se unen para formar el índice de fila como en el tipo «diccionario de series».
Lista de diccionarios o series	Cada elemento se convierte en una fila en el dataframe; las uniones de claves de diccionario o índices de series se convierten en las etiquetas de columna del dataframe.
Lista de listas o tuplas	Tratada como en el tipo «ndarray de dos dimensiones».
Otro dataframe	Se utilizan los índices del dataframe, a menos que se pasen otros distintos.
MaskedArray de NumPy	Igual que el tipo «ndarray de dos dimensiones», excepto que los valores enmascarados faltan en el resultado del dataframe.

Si los `index` y `columns` de un dataframe tienen establecido su atributo `name`, también se mostrará para cada uno:

```
In [79]: frame3.index.name = "year"
```

```
In [80]: frame3.columns.name = "state"
```

```
In [81]: frame3
```

```
Out[81]:
```

	state	Ohio	Nevada
year			
2000		1.5	NaN
2001		1.7	2.4
2002		3.6	2.9

A diferencia de las series, los dataframes no tienen atributo `name`. Su método `to_numpy` devuelve los datos contenidos en él como un `ndarray` bidimensional:

```
In [82]: frame3.to_numpy()
```

```
Out[82]:
```

```
array([[1.5, nan],
       [1.7, 2.4],
       [3.6, 2.9]])
```

Si las columnas del dataframe tienen tipos de datos distintos, se elegirá el tipo de datos del array devuelto para que acomode todas las columnas:

```
In [83]: frame2.to_numpy()
Out[83]:
array([[2000, 'Ohio', 1.5, nan],
       [2001, 'Ohio', 1.7, nan],
       [2002, 'Ohio', 3.6, nan],
       [2001, 'Nevada', 2.4, nan],
       [2002, 'Nevada', 2.9, nan],
       [2003, 'Nevada', 3.2, nan]], dtype=object)
```

Objetos índice

Los objetos índice de pandas son responsables de albergar etiquetas de ejes (incluyendo los nombres de columnas de un dataframe) y otros metadatos (como el nombre o los nombres de los ejes). Cualquier array u otra secuencia de etiquetas que se utilice al construir una serie o un dataframe se convierte internamente en un índice:

```
In [84]: obj = pd.Series(np.arange(3), index=["a", "b", "c"])
```

```
In [85]: index = obj.index
```

```
In [86]: index
```

```
Out[86]: Index(['a', 'b', 'c'], dtype='object')
```

```
In [87]: index[1:]
```

```
Out[87]: Index(['b', 'c'], dtype='object')
```

Los objetos índice son inmutables, y por eso no pueden ser modificados por el usuario:

```
index[1] = "d" # TypeError
```

La inmutabilidad hace más segura la conversión de objetos índice entre estructuras de datos:

```
In [88]: labels = pd.Index(np.arange(3))
```

```
In [89]: labels
Out[89]: Int64Index([0, 1, 2], dtype='int64')

In [90]: obj2 = pd.Series([1.5, -2.5, 0], index=labels)
```

```
In [91]: obj2
Out[91]:
0          1.5
1         -2.5
2          0.0
dtype: float64
```

```
In [92]: obj2.index is labels
Out[92]: True
```



Algunos usuarios no aprovecharán demasiado las capacidades ofrecidas por un objeto índice, pero como algunas operaciones producen resultados que contienen datos indexados, es importante comprender cómo funcionan.

Además de ser de estilo array, los índices se comportan también como un conjunto de tamaño fijo:

```
In [93]: frame3
Out[93]:
```

state	Ohio	Nevada
year		
2000	1.5	NaN
2001	1.7	2.4
2002	3.6	2.9

```
In [94]: frame3.columns
Out[94]: Index(['Ohio', 'Nevada'], dtype='object',
name='state')
```

```
In [95]: "Ohio" in frame3.columns
Out[95]: True
```

```
In [96]: 2003 in frame3.index
Out[96]: False
```

A diferencia de los conjuntos de Python, un índice de pandas puede contener etiquetas duplicadas:

```
In [97]: pd.Index(["foo", "foo", "bar", "bar"])
Out[97]: Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

Al realizar selecciones con etiquetas duplicadas se incluirán en la selección todas las apariciones de dicha etiqueta.

Cada índice dispone de diversos métodos y propiedades para lógica de conjuntos, que responden a otras preguntas habituales sobre los datos que contiene. En la tabla 5.2 se resumen las más útiles.

Tabla 5.2. Algunos métodos y propiedades del objeto índice.

Método/propiedad	Descripción
<code>append()</code>	Concatena con objetos índice adicionales, produciendo un nuevo índice.
<code>difference()</code>	Calcula la diferencia de conjuntos como un índice.
<code>intersection()</code>	Calcula la intersección de conjuntos.
<code>union()</code>	Calcula la unión de conjuntos.
<code>isin()</code>	Calcula un array booleano indicando si cada valor está contenido en la colección pasada.
<code>delete()</code>	Calcula un nuevo índice con el elemento del índice <i>i</i> borrado.
<code>drop()</code>	Calcula un nuevo índice borrando los valores pasados.
<code>insert()</code>	Calcula un nuevo índice insertando un elemento en el índice <i>i</i> .
<code>is_monotonic</code>	Devuelve True si cada elemento es mayor o igual que el elemento anterior.
<code>is_unique</code>	Devuelve True si el índice no tiene valores duplicados.
<code>unique()</code>	Calcula el array de valores únicos del índice.

5.2 Funcionalidad esencial

Esta sección nos guiará por la mecánica fundamental de la interacción con los datos contenidos en una serie o un dataframe. En los capítulos siguientes, profundizaremos en temas de análisis y manipulación de datos utilizando pandas. Este libro no está destinado a servir como documentación exhaustiva para la librería pandas; lo que haremos en realidad es centrarnos en que el usuario se familiarice con las funciones más utilizadas, dejando que aprenda las menos comunes mediante la documentación en línea de la herramienta.

Reindexación

Un método importante de los objetos pandas es la reindexación, que significa crear un nuevo objeto con los valores reordenados para que se alineen con el nuevo índice. Veamos un ejemplo:

```
In [98]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=["d",  
"b", "a", "c"])
```

```
In [99]: obj
```

```
Out[99]:
```

```
d                4.5  
b                7.2  
a               -5.3  
c                3.6  
dtype: float64
```

Llamar a `reindex` en esta serie reordena los datos según el nuevo índice, introduciendo los valores faltantes si algunos valores de índice no estaban ya presentes:

```
In [100]: obj2 = obj.reindex(["a", "b", "c", "d", "e"])
```

```
In [101]: obj2
```

```
Out[101]:
```

```
a                -5.3  
b                7.2  
c                3.6  
d                4.5
```



```
e
dtype: float64
```

NaN

Para series ordenadas, como, por ejemplo, las series temporales, quizá sea más interesante interpolar o rellenar con valores al reindexar. La opción `method` nos permite hacer esto, utilizando un método como `ffill`, que rellena hacia delante los valores:

```
In [102]: obj3 = pd.Series(["blue", "purple", "yellow"],
index=[0, 2, 4])
In [103]: obj3
Out[103]:
```

```
0
2
4
dtype: object
```

blue
purple
yellow

```
In [104]: obj3.reindex(np.arange(6), method="ffill")
Out[104]:
```

```
0
1
2
3
4
5
dtype: object
```

blue
blue
purple
purple
yellow
yellow

Con objetos `DataFrame`, `reindex` puede alterar el índice (fila), las columnas o ambas cosas. Cuando se pasa solo una secuencia, reindexa las filas en el resultado:

```
In [105]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
```

```
.....:      index=["a", "c", "d"],
.....:      columns=["Ohio", "Texas", "California"])
```

```
In [106]: frame
Out[106]:
```

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

```
In [107]: frame2 = frame.reindex(index=["a", "b", "c", "d"])
In [108]: frame2
Out[108]:
```

	Ohio	Texas	California
a	0.0	1.0	2.0
b	NaN	NaN	NaN
c	3.0	4.0	5.0
d	6.0	7.0	8.0

Las columnas se pueden reindexar con la palabra clave columns:

```
In [109]: states = ["Texas", "Utah", "California"]
In [110]: frame.reindex(columns=states)
Out[110]:
```

	Texas	Utah	California
a	1	NaN	2
c	4	NaN	5
d	7	NaN	8

Como "Ohio" no estaba en states, los datos de esa columna quedan fuera del resultado.

Otra forma de reindexar un determinado eje es pasar las etiquetas del nuevo eje como un argumento posicional y especificar después el eje para que se reindexe con la palabra clave axis:

```
In [111]: frame.reindex(states, axis="columns")
Out[111]:
```

	Texas	Utah	California
a			2

	1	NaN	
c	4	NaN	5
d	7	NaN	8

Véase la tabla 5.3 para más información sobre los argumentos de `reindex`.

Tabla 5.3. Argumentos de la función `reindex`.

Argumento	Descripción
<code>labels</code>	Nueva secuencia a utilizar como índice. Puede ser una instancia del índice o cualquier otra estructura de datos Python de tipo secuencia. Un índice se utilizará exactamente como tal, sin realizar copia alguna.
<code>index</code>	Usa la secuencia pasada como nuevas etiquetas de índice.
<code>columns</code>	Usa la secuencia pasada como nuevas etiquetas de columna.
<code>axis</code>	El eje a reindexar, ya sea "index" (filas) o "columns". El valor predeterminado es "index". Como alternativa se puede hacer <code>reindex(index=new_labels)</code> o <code>reindex(columns=new_labels)</code> .
<code>method</code>	Método de interpolación (relleno): "ffill" rellena hacia delante, mientras que "bfill" rellena hacia atrás.
<code>fill_value</code>	Sustituye el valor a utilizar al introducir datos faltantes reindexando. Utilizamos <code>fill_value="missing"</code> (el comportamiento predeterminado) si queremos que las etiquetas ausentes tengan valores nulos en el resultado.
<code>limit</code>	Cuando se rellena hacia delante o hacia atrás, el hueco de tamaño máximo (en número de elementos) a rellenar.
<code>tolerance</code>	Cuando se rellena hacia delante o hacia atrás, el hueco de tamaño máximo (en distancia numérica absoluta) a rellenar para coincidencias inexactas.
<code>level</code>	Coincide con el índice sencillo a nivel del índice jerárquico (MultiIndex); en caso contrario selecciona un subconjunto.
<code>copy</code>	Si es <code>True</code> , copia siempre los datos subyacentes incluso aunque el nuevo índice sea equivalente al antiguo; si es <code>False</code> , no copia los datos cuando los índices son equivalentes.

Como veremos posteriormente en el apartado «Selección en dataframes con loc e iloc», también se puede reindexar utilizando el operador loc, y muchos usuarios prefieren hacer esto siempre de esta forma. Esto funciona solamente si todas las etiquetas del nuevo índice ya existen en el dataframe (mientras que reindex insertará datos faltantes para etiquetas nuevas):

```
In [112]: frame.loc[["a", "d", "c"], ["California",  
"Texas"]]  
Out[112]:
```

	California	Texas
a	2	1
d	8	7
c	5	4

Eliminar entradas de un eje

Eliminar una o varias entradas de un eje es fácil si ya se dispone de un array índice o lista sin dichas entradas, ya que se puede usar el método reindex o la indexación basada en .loc. Como esto puede requerir proceso de datos y lógica de conjuntos, el método drop devolverá un nuevo objeto con el valor o valores indicados borrados de un eje:

```
In [113]: obj = pd.Series(np.arange(5.), index=["a", "b",  
"c", "d", "e"])
```

```
In [114]: obj
```

```
Out[114]:
```

a	0.0
b	1.0
c	2.0
d	3.0
e	4.0

dtype: float64

```
In [115]: new_obj = obj.drop("c")
```

```
In [116]: new_obj
```

```
Out[116]:
a                                0.0
b                                1.0
d                                3.0
e                                4.0
dtype: float64
```

```
In [117]: obj.drop(["d", "c"])
Out[117]:
a                                0.0
b                                1.0
e                                4.0
dtype: float64
```

Con objetos DataFrame, los valores de índice se pueden borrar de cualquier eje. Para ilustrar esto, primero creamos un dataframe de ejemplo:

```
In [118]: data = pd.DataFrame(np.arange(16).reshape((4,
4)),
.....:      index=["Ohio", "Colorado", "Utah", "New York"],
.....:      columns=["one", "two", "three", "four"])
```

```
In [119]: data
Out[119]:
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Llamar a drop con una secuencia de etiquetas eliminará valores de las etiquetas de fila (eje 0):

```
In [120]: data.drop(index=["Colorado", "Ohio"])
Out[120]:
```

	one	two	three	four
--	-----	-----	-------	------

Utah	8	9	10	11
New York	12	13	14	15

Para eliminar etiquetas de las columnas, usamos sin embargo la palabra clave `columns`:

```
In [121]: data.drop(columns=["two"])
Out[121]:
```

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

También se pueden quitar valores de las columnas pasando `axis=1` (como en NumPy) o `axis="columns"`:

```
In [122]: data.drop("two", axis=1)
Out[122]:
```

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

```
In [123]: data.drop(["two", "four"], axis="columns")
Out[123]:
```

	one	three
Ohio	0	2
Colorado	4	6
Utah	8	10
New York	12	14

Indexación, selección y filtrado

La indexación de series (`obj[...]`) funciona de manera análoga a la indexación de arrays NumPy, excepto que se pueden utilizar los valores de índice de la serie en lugar de solamente enteros. Aquí tenemos algunos ejemplos:

```
In [124]: obj = pd.Series(np.arange(4.), index=["a", "b",  
"c", "d"])
```

```
In [125]: obj
```

```
Out[125]:
```

a	0.0
b	1.0
c	2.0
d	3.0

```
dtype: float64
```

```
In [126]: obj["b"]
```

```
Out[126]: 1.0
```

```
In [127]: obj[1]
```

```
Out[127]: 1.0
```

```
In [128]: obj[2:4]
```

```
Out[128]:
```

c	2.0
d	3.0

```
dtype: float64
```

```
In [129]: obj[["b", "a", "d"]]
```

```
Out[129]:
```

b	1.0
a	0.0
d	3.0

```
dtype: float64
```

```
In [130]: obj[[1, 3]]
```

```
Out[130]:
```

b	1.0
---	-----

```
d                                     3.0
dtype: float64
```

```
In [131]: obj[obj < 2]
```

```
Out[131]:
a                                     0.0
b                                     1.0
dtype: float64
```

Aunque se pueden elegir datos por etiqueta de esta forma, el modo preferido para seleccionar valores de índice es mediante el operador especial `loc`:

```
In [132]: obj.loc[["b", "a", "d"]]
```

```
Out[132]:
b                                     1.0
a                                     0.0
d                                     3.0
dtype: float64
```

La razón para preferir `loc` es por el distinto tratamiento de los enteros cuando se indexan con `[]`. La indexación normal basada en `[]` tratará los enteros como etiquetas si el índice contiene enteros, de modo que el comportamiento difiere dependiendo del tipo de datos del índice. Por ejemplo:

```
In [133]: obj1 = pd.Series([1, 2, 3], index=[2, 0, 1])
```

```
In [134]: obj2 = pd.Series([1, 2, 3], index=["a", "b", "c"])
```

```
In [135]: obj1
```

```
Out[135]:
2                                     1
0                                     2
1                                     3
dtype: int64
```

```
In [136]: obj2
```



```
Out[136]:
a                                     1
b                                     2
c                                     3
dtype: int64
```

```
In [137]: obj1[[0, 1, 2]]
Out[137]:
0                                     2
1                                     3
2                                     1
dtype: int64
```

```
In [138]: obj2[[0, 1, 2]]
Out[138]:
a                                     1
b                                     2
c                                     3
dtype: int64
```

Cuando se utiliza `loc`, la expresión `obj.loc[[0, 1, 2]]` fallará cuando el índice no contiene enteros:

```
In [134]: obj2.loc[[0, 1]]
_____
KeyError                                Traceback (most recent call last)
/tmp/ipykernel_804589/4185657903.py in <module>
--> 1 obj2.loc[[0, 1]]
^ LONG EXCEPTION ABBREVIATED ^
KeyError: "None of [Int64Index([0, 1], dtype='int64')] are
in the [index]"
```

Como el operador `loc` indexa exclusivamente con etiquetas, hay también un operador `iloc` que indexa exclusivamente con enteros para trabajar de forma consistente, contenga o no enteros el índice:

```
In [139]: obj1.iloc[[0, 1, 2]]
Out[139]:
```

```

2
0
1
dtype: int64

```

1
2
3

```

In [140]: obj2.iloc[[0, 1, 2]]
Out[140]:
a
b
c
dtype: int64

```



También se puede cortar con etiquetas, pero funciona de un modo distinto al corte normal de Python en que el punto final es inclusivo:

```

In [141]: obj2.loc["b":"c"]
Out[141]:
b 2
c 3
dtype: int64

```

Asignar valores utilizando estos métodos modifica la sección correspondiente de la serie:

```
In [142]: obj2.loc["b":"c"] = 5
```

```

In [143]: obj2
Out[143]:
a
b
c
dtype: int64

```

1
5
5



Puede ser un error habitual de principiante intentar llamar a `loc` o `iloc` como funciones en lugar de «indexar dentro de» ellas con corchetes. La notación de corchetes se utiliza para habilitar las operaciones de corte y permitir la indexación en varios ejes con objetos `DataFrame`.

Indexar en un dataframe recupera una o varias columnas con un solo valor o una secuencia:

```
In [144]: data = pd.DataFrame(np.arange(16).reshape((4,
4)),
.....:    index=["Ohio", "Colorado", "Utah", "New York"],
.....:    columns=["one", "two", "three", "four"])
```

```
In [145]: data
Out[145]:
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

```
In [146]: data["two"]
```

```
Out[146]:
```

Ohio	1
Colorado	5
Utah	9
New York	13

```
Name: two, dtype: int64
```

```
In [147]: data[["three", "one"]]
```

```
Out[147]:
```

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12

Una indexación como esta tiene varios casos especiales. El primero es cortar o seleccionar datos con un array booleano:

```
In [148]: data[:2]
Out[148]:
```

	one	two	three	four
Ohio	0	1	2	3

Colorado	4	5	6	7
----------	---	---	---	---

```
In [149]: data[data["three"] > 5]
Out[149]:
```

	one	two	three	four
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

La sintaxis de selección de filas `data[:2]` se ofrece por comodidad. Pasar un solo elemento o una lista al operador `[]` selecciona las columnas.

Otra situación de uso es indexar con un dataframe booleano, como el producido por una comparación de escalares. Veamos un dataframe con todos los valores booleanos producidos por la comparación con un valor escalar:

```
In [150]: data < 5
Out[150]:
```

	one	two	three	four
Ohio	True	True	True	True
Colorado	True	False	False	False
Utah	False	False	False	False
New York	False	False	False	False

Podemos usar este dataframe para asignar el valor 0 a cada ubicación con el valor True, del siguiente modo:

```
In [151]: data[data < 5] = 0
```

```
In [152]: data
Out[152]:
```

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7

Utah	8	9	10	11
New York	12	13	14	15

Selección en dataframes con loc e iloc

Al igual que con el objeto Series, el objeto DataFrame dispone de los atributos especiales `loc` e `iloc` para la indexación basada en etiquetas y basada en enteros, respectivamente. Como los objetos DataFrame son bidimensionales, se puede seleccionar un subconjunto de las filas y columnas con notación de estilo NumPy utilizando etiquetas de ejes (`loc`) o enteros (`iloc`).

Como primer ejemplo, seleccionemos una sola fila por su etiqueta:

```
In [153]: data
Out[153]:
```

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

```
In [154]: data.loc["Colorado"]
Out[154]:
```

```
one                0
two                5
three             6
four              7
Name: Colorado, dtype: int64
```

El resultado de esto es una serie, con un índice que contiene las etiquetas de columna del dataframe. Para seleccionar varios roles creando un nuevo dataframe, pasamos una secuencia de etiquetas:

```
In [155]: data.loc[["Colorado", "New York"]]
Out[155]:
```

	one	two	three	four
Colorado	0	5	6	7
New York	12	13	14	15

Se puede combinar la selección de fila y columna en loc separando las selecciones con una coma:

```
In [156]: data.loc["Colorado", ["two", "three"]]
Out[156]:
two                5
three              6
Name: Colorado, dtype: int64
```

Después realizaremos algunas selecciones similares con enteros utilizando iloc:

```
In [157]: data.iloc[2]
Out[157]:
one                8
two                9
three             10
four             11
Name: Utah, dtype: int64
```

```
In [158]: data.iloc[[2, 1]]
Out[158]:
```

	one	two	three	four
Utah	8	9	10	11
Colorado	0	5	6	7

```
In [159]: data.iloc[2, [3, 0, 1]]
Out[159]:
four                11
one                 8
two                 9
Name: Utah, dtype: int64
```

```
In [160]: data.iloc[[1, 2], [3, 0, 1]]
Out[160]:
```

	four	one	two
Colorado	7	0	5
Utah	11	8	9

Ambas funciones de indexación funcionan con segmentos además de con etiquetas individuales o listas de etiquetas:

```
In [161]: data.loc["Utah", "two"]
Out[161]:
```

Ohio	0
Colorado	5
Utah	9

Name: two, dtype: int64

```
In [162]: data.iloc[:, :3][data.three > 5]
Out[162]:
```

	one	two	three
Colorado	0	5	6
Utah	8	9	10
New York	12	13	14

Los arrays booleanos se pueden utilizar con loc pero no con iloc:

```
In [163]: data.loc[data.three >= 2]
Out[163]:
```

	one	two	three	four
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Hay muchas formas de seleccionar y reordenar los datos contenidos en un objeto pandas. Para el caso de los dataframes, la tabla 5.4 proporciona un breve resumen de muchas de ellas. Como veremos después, hay distintas opciones adicionales para trabajar con índices jerárquicos.

Tabla 5.4. Opciones de indexado con objetos DataFrame.

Tipo	Notas
<code>df[column]</code>	Selecciona una sola columna o una secuencia de columnas del dataframe; casos especiales: array booleano (filtrado de filas), corte (segmentado de filas) o dataframe booleano (valores de conjunto basados en el mismo criterio).
<code>df.loc[rows]</code>	Selecciona una sola fila o un subconjunto de filas del dataframe por etiqueta.
<code>df.loc[:, cols]</code>	Selecciona una sola columna o un subconjunto de columnas por etiqueta.
<code>df.loc[rows, cols]</code>	Selecciona una fila (o filas) y una columna (o columnas) por etiqueta.
<code>df.iloc[rows]</code>	Selecciona una sola fila o un subconjunto de filas del dataframe por posición de entero.
<code>df.iloc[:, cols]</code>	Selecciona una sola columna o un subconjunto de columnas por posición de entero.
<code>df.iloc[rows, cols]</code>	Selecciona una fila (o filas) y una columna (o columnas) por posición de entero.
<code>df.at[row, col]</code>	Selecciona un solo valor escalar por etiqueta de fila y columna.
<code>df.iat[row, col]</code>	Selecciona un solo valor escalar por posición de fila y columna (enteros).
método <code>reindex</code>	Selecciona filas o columnas por etiquetas.

Inconvenientes de la indexación de enteros

Trabajar con objetos pandas indexados por enteros puede ser un obstáculo para nuevos usuarios, puesto que funcionan de forma diferente a

las estructuras de datos integradas de Python, como listas y tuplas. Por ejemplo, quizá uno no espera que el siguiente código genere un error:

```
In [164]: ser = pd.Series(np.arange(3.))
```

```
In [165]: ser
```

```
Out[165]:
```

```
0          0.0
1          1.0
2          2.0
dtype: float64
```

```
In [166]: ser[-1]
```

```
ValueError      Traceback (most recent call last)
/miniconda/envs/book-env/lib/python3.10/site-
packages/pandas/core/indexes/range.py in get_loc(self, key,
method, tolerance)
```

```
    384         try:
>    385         return self._range.index(new_key)
    386         except ValueError as err:
```

```
ValueError: -1 is not in range
```

The above exception was the direct cause of the following exception:

```
KeyError        Traceback (most recent call last)
```

```
<ipython-input-166-44969a759c20> in <module>
```

```
→ 1 ser[-1]
```

```
/miniconda/envs/book-env/lib/python3.10/site-
packages/pandas/core/series.py in __getitem__(self, key)
```

```
    956
    957     elif key_is_scalar:
>    958     return self._get_value(key)
```

```
    959
    960     if is_hashable(key):
/miniconda/envs/book-env/lib/python3.10/site-
packages/pandas/core/series.py in _get_value(self, label,
takeable)
```

```
    1067
```

```

1068     # Similar a Index.get_value, pero no volvemos
      a caer en posicional
-
1069     loc = self.index.get_loc(label)
>
1070     return self.index._get_values_for_loc(self,
      loc, label)
1071
/miniconda/envs/book-env/lib/python3.10/site-
packages/pandas/core/indexes/range.py in get_loc(self, key,
method, tolerance)
385     return self._range.index(new_key)
386     except ValueError as err:
-
387     raise KeyError(key) from err
>
388     self._check_indexing_error(key)
389     raise KeyError(key)
KeyError: -1

```

En este caso, pandas podría «retroceder» a la indexación de enteros, pero es difícil hacer esto en general sin introducir sutiles errores en el código del usuario. Aquí tenemos un índice que contiene 0, 1 y 2, pero pandas no quiere adivinar lo que quiere el usuario (indexación basada en etiquetas o en la posición):

```

In [167]: ser
Out[167]:

```

```

0                                0.0
1                                1.0
2                                2.0
dtype: float64

```

Por otro lado, con un índice no entero, no hay ambigüedad posible:

```

In [168]: ser2 = pd.Series(np.arange(3.), index=["a", "b",
"c"])

In [169]: ser2[-1]
Out[169]: 2.0

```

Si tenemos un índice de eje que contiene enteros, la selección de datos siempre estará orientada a las etiquetas. Como ya he dicho anteriormente, si se utiliza `loc` (para las etiquetas) o `iloc` (para los enteros), se obtiene exactamente lo que se desea:

```
In [170]: ser.iloc[-1]
Out[170]: 2.0
```

Por otra parte, la segmentación con enteros está siempre orientada a enteros:

```
In [171]: ser[:2]
Out[171]:
```

0	0.0
1	1.0

dtype: float64

Como resultado de estos escollos, es mejor preferir siempre indexar con `loc` e `iloc` para evitar ambigüedades.

Inconvenientes de la indexación encadenada

En la sección anterior vimos cómo se podían realizar selecciones flexibles en un dataframe con `loc` e `iloc`. Estos atributos de indexación pueden utilizarse también para modificar objetos `DataFrame` en el momento, pero hacerlo requiere un poco de cuidado.

Por ejemplo, en el dataframe de ejemplo anterior, podemos asignar a una columna o fila por etiqueta o posición de entero:

```
In [172]: data.loc[:, "one"] = 1
```

```
In [173]: data
Out[173]:
```

	one	two	three	four
Ohio	1	0	0	0
Colorado	1	5	6	7

Utah	1	9	10	11
New York	1	13	14	15

```
In [174]: data.iloc[2] = 5
```

```
In [175]: data
```

```
Out[175]:
```

	one	two	three	four
Ohio	1	0	0	0
Colorado	1	5	6	7
Utah	5	5	5	5
New York	1	13	14	15

```
In [176]: data.loc[data["four"] > 5] = 3
```

```
In [177]: data
```

```
Out[177]:
```

	one	two	three	four
Ohio	1	0	0	0
Colorado	3	3	3	3
Utah	5	5	5	5
New York	3	3	3	3

Un problema habitual para los nuevos usuarios de pandas es encadenar selecciones al asignar, como por ejemplo aquí:

```
In [177]: data.loc[data.three == 5]["three"] = 6
<ipython-input-11-0ed1cf2155d5>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a
DataFrame.
```

Try using `.loc[row_indexer,col_indexer] = value` instead

Dependiendo del contenido de los datos, esto podría imprimir un aviso `SettingWithCopyWarning` especial, que indica que se está intentando modificar un valor temporal (el resultado no vacío de

`data.loc[data.three == 5])` en lugar de los datos originales del dataframe, que podría ser el objetivo inicial. Aquí, `data` no se había modificado:

```
In [179]: data
Out[179]:
```

	one	two	three	four
Ohio	1	0	0	0
Colorado	3	3	3	3
Utah	5	5	5	5
New York	3	3	3	3

En estas situaciones, la solución es reescribir la asignación encadenada para utilizar una sola operación `loc`:

```
In [180]: data.loc[data.three == 5, "three"] = 6
```

```
In [181]: data
Out[181]:
```

	one	two	three	four
Ohio	1	0	0	0
Colorado	3	3	3	3
Utah	5	5	6	5
New York	3	3	3	3

Una buena regla general es evitar la indexación encadenada al realizar asignaciones. Existen otros casos en los que pandas generará `SettingWithCopyWarning` relacionados con la indexación encadenada. Le remito a este tema en la documentación en línea de pandas.

Aritmética y alineación de datos

Gracias a pandas se puede simplificar mucho el trabajo con objetos que tienen distintos índices. Por ejemplo, cuando se suman objetos, si algún par

de índices no es igual, el índice respectivo del resultado será la unión de los pares de índices. Veamos un ejemplo:

```
In [182]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=["a",  
"c", "d", "e"])
```

```
In [183]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],  
.....:      index=["a", "c", "e", "f", "g"])
```

```
In [184]: s1
```

```
Out[184]:
```

a	7.3
c	-2.5
d	3.4
e	1.5

dtype: float64

```
In [185]: s2
```

```
Out[185]:
```

a	-2.1
c	3.6
e	-1.5
f	4.0
g	3.1

dtype: float64

Sumando estos productos:

```
In [186]: s1 + s2
```

```
Out[186]:
```

a	5.2
c	1.1
d	NaN
e	0.0
f	NaN
g	NaN

dtype: float64

La alineación interna de datos introduce valores faltantes en las ubicaciones de etiquetas que no se superponen. Los valores faltantes se propagarán entonces en cálculos aritméticos posteriores.

En el caso de los objetos DataFrame, la alineación se realiza en filas y columnas:

```
In [187]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)),
.....:      columns=list("bcd"),
.....:      index=["Ohio", "Texas", "Colorado"])
```

```
In [188]: df2 = pd.DataFrame(np.arange(12.).reshape((4,
.....:      3)), columns=list("bde"),
.....:      index=["Utah", "Ohio", "Texas", "Oregon"])
```

```
In [189]: df1
Out[189]:
```

	b	c	d
Ohio	0.0	1.0	2.0
Texas	3.0	4.0	5.0
Colorado	6.0	7.0	8.0

```
In [190]: df2
Out[190]:
```

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

Sumar ambos devuelve un dataframe con un índice y columnas que son las uniones de las correspondientes de cada dataframe:

```
In [191]: df1 + df2
Out[191]:
```

	b	c	d	e
Colorado	NaN	NaN	NaN	NaN
Ohio	3.0	NaN	6.0	NaN
Oregon	NaN	NaN	NaN	NaN
Texas	9.0	NaN	12.0	NaN
Utah	NaN	NaN	NaN	NaN

Como las columnas "c" y "e" no están en ambos objetos DataFrame, aparecen como ausentes en el resultado. Lo mismo ocurre con las filas con etiquetas que no son comunes para ambos objetos.

Si se suman objetos DataFrame sin etiquetas de columna o fila en común, el resultado contendrá todo valores nulos:

```
In [192]: df1 = pd.DataFrame({"A": [1, 2]})
```

```
In [193]: df2 = pd.DataFrame({"B": [3, 4]})
```

```
In [194]: df1
```

```
Out[194]:
```

	A
0	1
1	2

```
In [195]: df2
```

```
Out[195]:
```

	B
0	3
1	4

```
In [196]: df1 + df2
```

```
Out[196]:
```

	A	B
0	NaN	NaN
1	NaN	NaN

Métodos aritméticos con valores de relleno

En operaciones aritméticas entre objetos indexados de forma diferente, quizá interese rellenar con un valor especial, como el cero, cuando se encuentra una etiqueta de eje en un objeto pero no en el otro. Aquí tenemos un ejemplo en el que fijamos un determinado valor a nulo asignándole `np.nan`:

```
In [197]: df1 = pd.DataFrame(np.arange(12.).reshape((3,
4)),
.....:                      columns=list("abcd"))
```

```
In [198]: df2 = pd.DataFrame(np.arange(20.).reshape((4,
5)),
.....:                      columns=list("abcde"))
```

```
In [199]: df2.loc[1, "b"] = np.nan
```

```
In [200]: df1
Out[200]:
```

	a	b	c	d
0	0.0	1.0	2.0	3.0
1	4.0	5.0	6.0	7.0
2	8.0	9.0	10.0	11.0

```
In [201]: df2
Out[201]:
```

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	NaN	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0

Sumarlos da como resultado valores ausentes en las ubicaciones que no se superponen:

```
In [202]: df1 + df2
Out[202]:
```

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	NaN
1	9.0	NaN	13.0	15.0	NaN
2	18.0	20.0	22.0	24.0	NaN
3	NaN	NaN	NaN	NaN	NaN

Utilizando el método `add` en `df1`, pasamos `df2` y un argumento a `fill_value`, que sustituye el valor pasado por cualquier valor faltante en la operación:

```
In [203]: df1.add(df2, fill_value=0)
Out[203]:
```

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	4.0
1	9.0	5.0	13.0	15.0	9.0
2	18.0	20.0	22.0	24.0	14.0
3	15.0	16.0	17.0	18.0	19.0

Véase en la tabla 5.5 un listado de métodos de series y dataframes para aritmética. Cada uno tiene un equivalente, empezando con la letra `r`, que tiene los argumentos invertidos. Por lo tanto estas dos sentencias son equivalentes:

```
In [204]: 1 / df1
Out[204]:
```

	a	b	c	d
0	inf	1.000000	0.500000	0.333333
1	0.250	0.200000	0.166667	0.142857
2	0.125	0.111111	0.100000	0.090909

```
In [205]: df1.rdiv(1)
Out[205]:
```

	a	b	c	d
0	inf	1.000000	0.500000	0.333333
1	0.250	0.200000	0.166667	0.142857
2	0.125	0.111111	0.100000	0.090909

Del mismo modo, al reindexar una serie o un dataframe se puede también especificar un valor de relleno diferente:

```
In [206]: df1.reindex(columns=df2.columns, fill_value=0)
Out[206]:
```

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	0
1	4.0	5.0	6.0	7.0	0
2	8.0	9.0	10.0	11.0	0

Tabla 5.5. Métodos aritméticos flexibles.

Método	Descripción
add, radd	Métodos para suma (+).
sub, rsub	Métodos para resta (-).
div, rdiv	Métodos para división (/).
floordiv, rfloordiv	Métodos para división de piso (/).
mul, rmul	Métodos para multiplicación (*).
pow, rpow	Métodos para exponenciación (**).

Operaciones entre objetos DataFrame y Series

Igual que con arrays NumPy de distintas dimensiones, la aritmética entre objetos DataFrame y Series está también definida. En primer lugar, y como ejemplo motivador, veamos la diferencia entre un array bidimensional y una de sus filas:

```
In [207]: arr = np.arange(12.).reshape((3, 4))
```

```
In [208]: arr
```

```
Out[208]:
```

```
array([[ 0.,  1.,  2.,  3.],  
  
       [ 4.,  5.,  6.,  7.],  
       [ 8.,  9., 10., 11.]])
```

```
In [209]: arr[0]
```

```
Out[209]: array([0.,  1.,  2.,  3.]])
```

```
In [210]: arr-arr[0]
```

```
Out[210]:
```

```
array([[0.,  0.,  0.,  0.],  
  
       [4.,  4.,  4.,  4.],  
       [8.,  8.,  8.,  8.]])
```

Cuando restamos `arr[0]` de `arr`, la resta se realiza una vez por cada fila. A esto se denomina difusión, y se explica con más detalle en el apéndice A, ya que tiene que ver con los arrays NumPy en general. Las operaciones entre un dataframe y una serie son similares:

```
In [211]: frame = pd.DataFrame(np.arange(12.).reshape((4,  
3))),
```

```
.....:     columns=list("bde"),
```

```
.....:     index=["Utah", "Ohio", "Texas", "Oregon"]])
```

```
In [212]: series = frame.iloc[0]
```

```
In [213]: frame
```

```
Out[213]:
```

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

```
In [214]: series
```

```
Out[214]:
```

```
b                                0.0
d                                1.0
e                                2.0
Name: Utah, dtype: float64
```

De forma predeterminada, la aritmética entre un dataframe y una serie hace coincidir el índice de la serie con las columnas del dataframe, difundiendo las filas:

```
In [215]: frame - series
```

```
Out[215]:
```

	b	d	e
Utah	0.0	0.0	0.0
Ohio	3.0	3.0	3.0
Texas	6.0	6.0	6.0
Oregon	9.0	9.0	9.0

Si el valor de un índice no se encuentra en las columnas del dataframe o en el índice de la serie, los objetos se reindexarán para formar la unión:

```
In [216]: series2 = pd.Series(np.arange(3), index=["b", "e", "f"])
```

```
In [217]: series2
```

```
Out[217]:
```

```
b                                0
e                                1
f                                2
dtype: int64
```

```
In [218]: frame + series2
```

```
Out[218]:
```

	b	d	e	f
--	---	---	---	---

Utah	0.0	NaN	3.0	NaN
Ohio	3.0	NaN	6.0	NaN
Texas	6.0	NaN	9.0	NaN
Oregon	9.0	NaN	12.0	NaN

Si se desea difundir por columnas, haciendo coincidir las filas, se debe utilizar uno de los métodos aritméticos y especificar que coincida con el índice. Por ejemplo:

```
In [219]: series3 = frame["d"]
```

```
In [220]: frame
```

```
Out[220]:
```

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

```
In [221]: series3
```

```
Out[221]:
```

Utah	1.0
Ohio	4.0
Texas	7.0
Oregon	10.0

Name: d, dtype: float64

```
In [222]: frame.sub(series3, axis="index")
```

```
Out[222]:
```

	b	d	e
Utah	-1.0	0.0	1.0
Ohio	-1.0	0.0	1.0
Texas	-1.0	0.0	1.0
Oregon	-1.0	0.0	1.0

El eje que se pasa es el eje con el que coincidir. En este caso queremos decir que hay que coincidir con el índice de fila del DataFrame (`axis="index"`) y difundir a lo largo de las columnas.

Aplicación y asignación de funciones

Las *ufuncs* de NumPy (métodos de array por elementos) trabajan también con objetos pandas:

```
In [223]: frame = pd.DataFrame(np.random.standard_normal((4,
3)),
```

```
.....:     columns=list("bde"),
.....:     index=["Utah", "Ohio", "Texas", "Oregon"])
```

```
In [224]: frame
Out[224]:
```

	b	d	e
Utah	-0.204708	0.478943	-0.519439
Ohio	-0.555730	1.965781	1.393406
Texas	0.092908	0.281746	0.769023
Oregon	1.246435	1.007189	-1.296221

```
In [225]: np.abs(frame)
Out[225]:
```

	b	d	e
Utah	0.204708	0.478943	0.519439
Ohio	0.555730	1.965781	1.393406
Texas	0.092908	0.281746	0.769023
Oregon	1.246435	1.007189	1.296221

Otra operación frecuente es aplicar una función en arrays unidimensionales a cada columna o fila. El método `apply` del objeto DataFrame hace exactamente esto:

```
In [226]: def f1(x):
.....:         return x.max()-x.min()
```

```
In [227]: frame.apply(f1)
Out[227]:
```

```
b          1.802165
d          1.684034
e          2.689627
dtype: float64
```

Aquí la función `f`, que calcula la diferencia entre el máximo y el mínimo de una serie, se invoca una vez para cada columna en `frame`. El resultado es que una serie tiene las columnas de `frame` como índice.

Si se pasa `axis="columns"` a `apply`, lo que ocurre es que la función se invoca una vez por fila. Una forma útil de pensar en esto es como si se «aplicara en todas las columnas»:

```
In [228]: frame.apply(f1, axis="columns")
Out[228]:
```

```
Utah          0.998382
Ohio          2.521511
Texas         0.676115
Oregon        2.542656
dtype: float64
```

Muchas de las estadísticas de array más comunes (como `sum` y `mean`) son métodos del objeto `DataFrame`, de modo que no es necesario utilizar `apply`.

La función pasada a `apply` no tiene que devolver un valor escalar; también puede devolver una serie con varios valores:

```
In [229]: def f2(x):
.....:         return pd.Series([x.min(), x.max()], index=
.....:                             ["min", "max"])
```

```
In [230]: frame.apply(f2)
```


Out[230]:

	b	d	e
min	-0.555730	0.281746	-1.296221
max	1.246435	1.965781	1.393406

También se pueden emplear funciones Python por elementos. Supongamos que queremos calcular una cadena de texto formateada a partir de cada valor de punto flotante en frame. Esto puede hacerse con `applymap`:

```
In [231]: def my_format(x):  
.....:         return f"{x:.2f}"
```

```
In [232]: frame.applymap(my_format)  
Out[232]:
```

	b	d	e
Utah	-0.20	0.48	-0.52
Ohio	-0.56	1.97	1.39
Texas	0.09	0.28	0.77
Oregon	1.25	1.01	-1.30

La razón del nombre `applymap` es que las series tienen un método `map` para aplicar una función por elementos:

```
In [233]: frame["e"].map(my_format)  
Out[233]:
```

Utah	-0.52
Ohio	1.39
Texas	0.77
Oregon	-1.30

Name: e, dtype: object

Ordenación y asignación de rangos

Ordenar un conjunto de datos según un cierto criterio es otra operación interna importante. Para ordenar lexicográficamente por etiqueta de fila o columna, se emplea el método `sort_index`, que devuelve un objeto nuevo y ordenado:

```
In [234]: obj = pd.Series(np.arange(4), index=["d", "a", "b", "c"])
```

```
In [235]: obj
```

```
Out[235]:
```

d	0
a	1
b	2
c	3

dtype: int64

```
In [236]: obj.sort_index()
```

```
Out[236]:
```

a	1
b	2
c	3
d	0

dtype: int64

Con un dataframe, se puede ordenar por el índice de cada eje:

```
In [237]: frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
.....:      index=["three", "one"],
.....:      columns=["d", "a", "b", "c"])
```

```
In [238]: frame
```

```
Out[238]:
```

	d	a	b	c
three	0	1	2	3
one	4	5	6	7

```
In [239]: frame.sort_index()
Out[239]:
```

	d	a	b	c
one	4	5	6	7
three	0	1	2	3

```
In [240]: frame.sort_index(axis="columns")
Out[240]:
```

	a	b	c	d
three	1	2	3	0
one	5	6	7	4

Los datos se colocan en orden ascendente de forma predeterminada, pero también se pueden organizar en orden descendente:

```
In [241]: frame.sort_index(axis="columns", ascending=False)
Out[241]:
```

	d	c	b	a
three	0	3	2	1
one	4	7	6	5

Para ordenar una serie por sus valores, empleamos su método `sort_values`:

```
In [242]: obj = pd.Series([4, 7, -3, 2])
```

```
In [243]: obj.sort_values()
Out[243]:
```

2	-3
3	2
0	4
1	7

dtype: int64

Los valores que puedan faltar se ordenan al final de la serie de forma predeterminada:

```
In [244]: obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])
In [245]: obj.sort_values()
Out[245]:
```

4	-3.0
5	2.0
0	4.0
2	7.0
1	NaN
3	NaN

```
dtype: float64
```

Pero dichos valores ausentes se pueden organizar al principio con la opción `na_position`:

```
In [246]: obj.sort_values(na_position="first")
Out[246]:
```

1	NaN
3	NaN
4	-3.0
5	2.0
0	4.0
2	7.0

```
dtype: float64
```

Al ordenar un dataframe, es posible emplear los datos de una o varias columnas como claves de ordenación. Para ello, pasamos uno o varios nombres de columna a `sort_values`:

```
In [247]: frame = pd.DataFrame({"b": [4, 7, -3, 2], "a": [0, 1, 0, 1]})
In [248]: frame
Out[248]:
```

	b	a
0	4	0
1	7	1
2	-3	0
3	2	1

```
In [249]: frame.sort_values("b")
Out[249]:
```

	b	a
2	-3	0
3	2	1
0	4	0
1	7	1

Para ordenar por varias columnas, pasamos una lista de nombres:

```
In [250]: frame.sort_values(["a", "b"])
Out[250]:
```

	b	a
2	3	0
0	4	0
3	2	1
1	7	1

La asignación de rangos hace lo propio, es decir, asigna rangos desde uno hasta el número de puntos de datos válidos de un array, empezando por el valor mínimo. Los métodos rank para series y dataframes son el punto de partida; por defecto, rank desempata asignando a cada grupo el rango medio:

```
In [251]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
```

```
In [252]: obj.rank()
Out[252]:
```

0	6.5
---	-----

1	1.0
2	6.5
3	4.5
4	3.0
5	2.0
6	4.5

dtype: float64

Los rangos también se pueden asignar según el orden en el que se observan en los datos:

```
In [253]: obj.rank(method="first")
Out[253]:
```

0	6.0
1	1.0
2	7.0
3	4.0
4	3.0
5	2.0
6	5.0

dtype: float64

En este caso, en lugar de usar el rango promedio 6.5 para las entradas 0 y 2, han sido fijadas en 6 y 7, porque la etiqueta 0 precede a la etiqueta 2 en los datos.

Se puede organizar también en orden descendente:

```
In [254]: obj.rank(ascending=False)
Out[254]:
```

0	1.5
1	7.0
2	1.5
3	3.5
4	5.0
5	6.0
6	3.5

dtype: float64

Véase en la tabla 5.6 una lista de métodos de desempate disponibles.

Tabla 5.6. Métodos de desempate con rank.

Método	Descripción
"average"	Valor predeterminado: asigna el rango medio a cada entrada del grupo empatado.
"min"	Utiliza el rango mínimo para el grupo completo.
"max"	Utiliza el rango máximo para el grupo completo.
"first"	Asigna rangos en el orden en que aparecen los valores en los datos.
"dense"	Igual que method="min", pero los rangos siempre aumentan en 1 entre grupos en lugar del número de elementos iguales en un grupo.

El objeto DataFrame permite calcular rangos a lo largo de las filas o las columnas.

```
In [255]: frame = pd.DataFrame({"b": [4.3, 7, -3, 2], "a":  
[0, 1, 0, 1],  
.....:                        "c": [-2, 5, 8, -2.5]})
```

```
In [256]: frame  
Out[256]:
```

	b	a	c
0	4.3	0	-2.0
1	7.0	1	5.0
2	-3.0	0	8.0
3	2.0	1	-2.5

```
In [257]: frame.rank(axis="columns")  
Out[257]:
```

	b	a	c
0	3.0	2.0	1.0
1	3.0	1.0	2.0
2	1.0	2.0	3.0

Índices de ejes con etiquetas duplicadas

Hasta ahora, casi todos los ejemplos que hemos visto tienen etiquetas de eje únicas (valores de índice). Aunque muchas funciones de pandas (como por ejemplo `reindex`) requieren que las etiquetas sean únicas, no es obligatorio. Veamos una serie pequeña con índices duplicados:

```
In [258]: obj = pd.Series(np.arange(5), index=["a", "a",
        "b", "b", "c"])
```

```
In [259]: obj
Out[259]:
```

```
a                                0
a                                1
b                                2
b                                3
c                                4
dtype: int64
```

La propiedad `is_unique` del índice puede indicar si sus etiquetas son únicas o no:

```
In [260]: obj.index.is_unique
Out[260]: False
```

La selección de datos es una de las características principales que se comporta de forma diferente con duplicados. Indexar una etiqueta con varias entradas devuelve una serie, mientras que las entradas únicas devuelven un valor escalar:

```
In [261]: obj["a"]
Out[261]:
```

```
a                                0
a                                1
dtype: int64
```



```
In [262]: obj["c"]
Out[262]: 4
```

Esto puede conseguir que el código se complique bastante, ya que el tipo de resultado de la indexación puede variar, dependiendo de si una etiqueta se repite o no. La misma lógica aplica a la indexación de filas (o columnas) en un dataframe:

```
In [263]: df = pd.DataFrame(np.random.standard_normal((5,
3)),
.....:                      index=["a", "a", "b", "b", "c"])
```

```
In [264]: df
Out[264]:
```

		0	1	2
a	0.274992	0.228913	1.352917	
a	0.886429	-2.001637	-0.371843	
b	1.669025	-0.438570	-0.539741	
b	0.476985	3.248944	-1.021228	
c	-0.577087	0.124121	0.302614	

```
In [265]: df.loc["b"]
Out[265]:
```

		0	1	2
b	1.669025	-0.438570	-0.539741	
b	0.476985	3.248944	-1.021228	

```
In [266]: df.loc["c"]
Out[266]:
```

0	-0.577087
1	0.124121
2	0.302614

Name: c, dtype: float64

5.3 Resumir y calcular estadísticas descriptivas

Los objetos pandas están equipados con un conjunto de métodos matemáticos y estadísticos comunes. La mayoría entran en la categoría de reducciones o estadísticas de resumen, es decir, métodos que extraen un solo valor (como la suma o el promedio) de una serie, o una serie de valores de las filas y columnas de un dataframe. Comparados con los métodos similares que se pueden encontrar en los arrays NumPy, incorporan manipulación de datos faltantes. Veamos este pequeño dataframe:

```
In [267]: df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],  
.....:                    [np.nan, np.nan], [0.75, -1.3]],  
.....:                    index=["a", "b", "c", "d"],  
.....:                    columns=["one", "two"])
```

```
In [268]: df  
Out[268]:
```

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

Llamar al método `sum` del dataframe devuelve una serie que contiene sumas de columna:

```
In [269]: df.sum()  
Out[269]:
```

one	9.25
two	-5.80
dtype:	float64

Sin embargo, pasar `axis="columns"` o `axis=1` suma en todas las columnas:

```
In [270]: df.sum(axis="columns")
Out[270]:
```

a	1.40
b	2.60
c	0.00
d	-0.55

dtype: float64

Cuando todos los valores de una fila o columna son nulos o faltan, la suma es 0, mientras que, si hay algún valor que no es nulo, entonces el resultado es no nulo o faltante. Esto se puede deshabilitar con la opción `skipna`, en cuyo caso cualquier valor nulo en una fila o columna asigna al resultado correspondiente el nombre de nulo o faltante:

```
In [271]: df.sum(axis="index", skipna=False)
Out[271]:
```

one	NaN
two	NaN

dtype: float64

```
In [272]: df.sum(axis="columns", skipna=False)
Out[272]:
```

a	NaN
b	2.60
c	NaN
d	-0.55

dtype: float64

Algunas agregaciones, como `mean`, requieren al menos un valor no nulo para producir un resultado con valor, así que aquí tenemos:

```
In [273]: df.mean(axis="columns")
Out[273]:
```

a	1.400
b	1.300

```

c
d
dtype: float64
NaN
-0.275

```

Véase en la tabla 5.7 una lista de opciones habituales para cada método de reducción.

Tabla 5.7. Opciones para métodos de reducción.

Método	Descripción
axis	Eje para reducir; "index" para las filas del dataframe y "columns" para las columnas.
skipna	Excluye los valores faltantes; True de forma predeterminada.
level	Reduce los agrupados por nivel si el eje se indexa de forma jerárquica (MultiIndex).

Algunos métodos, como `idxmin` e `idxmax`, devuelven estadísticas indirectas, como el valor de índice en el que se alcanzan los valores mínimo o máximo:

```

In [274]: df.idxmax()
Out[274]:

```

```

one
two
dtype: object
b
d

```

Otros métodos son acumulaciones:

```

In [275]: df.cumsum()
Out[275]:

```

```

one
two
a      1.40      NaN
b      8.50     -4.5
c       NaN      NaN
d      9.25     -5.8

```

Algunos métodos no son ni reducciones ni acumulaciones. Un ejemplo es describe, dado que produce varias estadísticas de resumen de una sola vez:

```
In [276]: df.describe()
Out[276]:
```

	one	two
count	3.000000	2.000000
mean	3.083333	-2.900000
std	3.493685	2.262742
min	0.750000	-4.500000
25%	1.075000	-3.700000
50%	1.400000	2.900000
75%	4.250000	-2.100000
max	7.100000	-1.300000

En datos no numéricos, describe produce estadísticas de resumen alternativas:

```
In [277]: obj = pd.Series(["a", "a", "b", "c"] * 4)
In [278]: obj.describe()
Out[278]:
```

count	16
unique	3
top	a
freq	8
dtype: object	

Véase en la tabla 5.8 una lista completa de los métodos de estadísticas de resumen y relacionados.

Tabla 5.8. Estadísticas descriptivas y de resumen.

Método	Descripción
--------	-------------

Método	Descripción
count	Número de valores que no son nulos.
describe	Calcula un conjunto de estadísticas de resumen.
min, max	Calcula los valores mínimo y máximo.
argmin, argmax	Calcula ubicaciones de índice (enteros) en las que se obtienen los valores mínimo o máximo, respectivamente; no está disponible con objetos Dataframe.
idxmin, idxmax	Calcula etiquetas de índice en las que se obtienen los valores mínimo o máximo, respectivamente.
quantile	Calcula el cuantil de muestra entre 0 y 1 (valor predeterminado: 0.5).
sum	Suma de valores.
mean	Promedio de valores.
median	Media aritmética (50 % cuantil) de valores.
mad	Desviación media absoluta del valor promedio.
prod	Producto de todos los valores.
var	Varianza de los valores de muestra.
std	Desviación estándar de los valores de muestra.
skew	Asimetría (tercer momento) de los valores de muestra.
kurt	Curtosis (cuarto momento) de los valores de muestra.
cumsum	Suma acumulada de los valores.
cummin, cummax	Mínimo o máximo acumulado de los valores, respectivamente.
cumprod	Producto acumulado de valores.
diff	Calcula la primera diferencia aritmética (útil para series temporales).
pct_change	Calcula cambios de porcentaje.

Correlación y covarianza

Algunas estadísticas de resumen, como la correlación y la covarianza, se calculan a partir de pares de argumentos. Supongamos algunos dataframes de precios y volúmenes de acciones, obtenidos originalmente de Yahoo! Finance y disponibles en archivos pickle binarios de Python, que se pueden encontrar en los conjuntos de datos que acompañan al libro:

```
In [279]: price = pd.read_pickle("examples/yahoo_price.pkl")
```

```
In [280]: volume = pd.read_pickle("examples/yahoo_volume.pkl")
```

Ahora calculamos cambios de porcentaje en los precios, una operación de serie temporal que exploraremos con más detalle en el capítulo 11:

```
In [281]: returns = price.pct_change()
```

```
In [282]: returns.tail()
```

```
Out[282]:
```

	AAPL	GOOG	IBM	MSFT
Date				
2016-10-17	-0.000680	0.001837	0.002072	-0.003483
2016-10-18	-0.000681	0.019616	-0.026168	0.007690
2016-10-19	-0.002979	0.007846	0.003583	-0.002255
2016-10-20	-0.000512	-0.005652	0.001719	-0.004867
2016-10-21	-0.003930	0.003011	-0.012474	0.042096

El método `corr` del objeto `Series` calcula la correlación de los valores superpuestos, no nulos y alineados por índice en dos series. De forma similar, `cov` calcula la covarianza:

```
In [283]: returns["MSFT"].corr(returns["IBM"])
```

```
Out[283]: 0.49976361144151144
```

```
In [284]: returns["MSFT"].cov(returns["IBM"])
```

```
Out[284]: 8.870655479703546e-05
```

Como `MSFT` es un nombre de variable Python válido, podemos también seleccionar estas columnas empleando una sintaxis más concisa:

```
In [285]: returns["MSFT"].corr(returns["IBM"])
Out[285]: 0.49976361144151144
```

Los métodos `corr` y `cov` del objeto `DataFrame`, por otro lado, devuelven una correlación completa o una matriz de covarianza como un dataframe, respectivamente:

```
In [286]: returns.corr()
Out[286]:
```

	AAPL	GOOG	IBM	MSFT
AAPL	1.000000	0.407919	0.386817	0.389695
GOOG	0.407919	1.000000	0.405099	0.465919
IBM	0.386817	0.405099	1.000000	0.499764
MSFT	0.389695	0.465919	0.499764	1.000000

```
In [287]: returns.cov()
Out[287]:
```

	AAPL	GOOG	IBM	MSFT
AAPL	0.000277	0.000107	0.000078	0.000095
GOOG	0.000107	0.000251	0.000078	0.000108
IBM	0.000078	0.000078	0.000146	0.000089
MSFT	0.000095	0.000108	0.000089	0.000215

Utilizando el método `corrwith` del objeto `DataFrame` se pueden calcular correlaciones por pares entre las columnas o filas de un dataframe con otra serie o dataframe. Pasar una serie devuelve otra con el valor de correlación calculado para cada columna:

```
In [288]: returns.corrwith(returns["IBM"])
Out[288]:
```

AAPL	0.386817
GOOG	0.405099
IBM	1.000000
MSFT	0.499764

dtype: float64

Pasar un dataframe calcula las correlaciones de los nombres de columna coincidentes. En este caso se han calculado las correlaciones de los cambios de porcentaje con volumen:

```
In [289]: returns.corrwith(volume)
Out[289]:
```

AAPL	-0.075565
GOOG	-0.007067
IBM	-0.204849
MSFT	-0.092950
dtype:	float64

Sin embargo, pasar `axis="columns"` hace las cosas fila a fila. En todos los casos, los puntos de datos se alinean por etiqueta antes de que se calcule la correlación.

Valores únicos, recuentos de valores y pertenencia

Otra clase de métodos asociados extrae información acerca de los valores contenidos en una serie unidimensional. Para ilustrarlos, veamos este ejemplo:

```
In [290]: obj = pd.Series(["c", "a", "d", "a", "a", "b",
                          "b", "c", "c"])
```

La primera función es `unique`, que proporciona un array con los valores únicos de una serie:

```
In [291]: uniques = obj.unique()

In [292]: uniques
Out[292]: array(['c', 'a', 'd', 'b'], dtype=object)
```

Los valores únicos no se devuelven necesariamente en el orden en el que primero aparecen, y tampoco ordenados, aunque podrían ordenarse a posteriori si fuera necesario (`uniques.sort()`). De forma similar, `value_counts` calcula una serie que contiene frecuencias de valores:

```
In [293]: obj.value_counts()
Out[293]:
```

```
c      3
a      3
b      2
d      1
dtype: int64
```

La serie se ordena por valor en orden descendente por comodidad. También está disponible `value_counts` como método pandas de nivel superior, que se puede emplear con arrays NumPy u otras secuencias de Python:

```
In [294]: pd.value_counts(obj.to_numpy(), sort=False)
Out[294]:
```

```
c      3
a      3
d      1
b      2
dtype: int64
```

`isin` realiza una comprobación de la pertenencia a un conjunto vectorizado y puede ser útil al filtrar un conjunto de datos para obtener un subconjunto de valores en una serie o una columna de un dataframe:

```
In [295]: obj
Out[295]:
```

```
0      c
1      a
2      d
3      a
4      a
5      b
6      b
7      c
8      c
```

```
dtype: object
```

```
In [296]: mask = obj.isin(["b", "c"])
```

```
In [297]: mask
```

```
Out[297]:
```

0	True
1	False
2	False
3	False
4	False
5	True
6	True
7	True
8	True

```
dtype: bool
```

```
In [298]: obj[mask]
```

```
Out[298]:
```

0	c
5	b
6	b
7	c
8	c

```
dtype: object
```

Relacionado con `isin` tenemos el método `Index.get_indexer`, que proporciona un array de índices a partir de otro array de valores posiblemente no diferenciados, para obtener otro array de valores diferentes:

```
In [299]: to_match = pd.Series(["c", "a", "b", "b", "c",  
                                "a"])
```

```
In [300]: unique_vals = pd.Series(["c", "b", "a"])
```

```
In [301]: indices =  
pd.Index(unique_vals).get_indexer(to_match)
```

```
In [302]: indices
Out[302]: array([0, 2, 1, 1, 0, 2])
```

Véase en la tabla 5.9 una referencia de estos métodos.

Tabla 5.9. Métodos únicos, de recuentos de valores y de pertenencia a conjuntos.

Método	Descripción
isin	Calcula un array booleano indicando si cada valor de una serie o un dataframe está contenido en la secuencia de valores pasada.
get_indexer	Calcula índices enteros para cada valor de un array para obtener otro array de valores diferentes; es útil para operaciones de alineación de datos y de tipo JOIN.
unique	Calcula un array de valores únicos de una serie, devueltos en el orden observado.
value_counts	Devuelve una serie que contiene valores únicos como índice y frecuencias como valores, un recuento ordenado en orden descendente.

En algunos casos, quizá interese calcular un histograma con varias columnas asociadas en un dataframe. Aquí tenemos un ejemplo:

```
In [303]: data = pd.DataFrame({"Qu1": [1, 3, 4, 3, 4],
.....:                        "Qu2": [2, 3, 1, 2, 3],
.....:                        "Qu3": [1, 5, 2, 4, 4]})
```

```
In [304]: data
Out[304]:
```

	Qu1	Qu2	Qu3
0	1	2	1
1	3	3	5
2	4	1	2
3	3	2	4
4	4	3	4

Podemos calcular los recuentos de valores para una sola columna, de este modo:

```
In [305]: data["Qu1"].value_counts().sort_index()
Out[305]:
```

1	1
3	2
4	2

```
Name: Qu1, dtype: int64
```

Para calcular esto para todas las columnas, pasamos `pandas.value_counts` al método `apply` del dataframe:

```
In [306]: result = data.apply(pd.value_counts).fillna(0)
```

```
In [307]: result
Out[307]:
```

	Qu1	Qu2	Qu3
1	1.0	1.0	1.0
2	0.0	2.0	1.0
3	2.0	2.0	0.0
4	2.0	0.0	2.0
5	0.0	0.0	1.0

En este caso, las etiquetas de fila del resultado son los distintos valores que ocurren en todas las columnas. Los valores son los recuentos respectivos de estos valores en cada columna.

Hay también un método `DataFrame.value_counts`, pero calcula los recuentos teniendo en cuenta cada fila del dataframe como una tupla, para determinar el número de apariciones de cada fila diferente:

```
In [308]: data = pd.DataFrame({"a": [1, 1, 1, 2, 2], "b": [0, 0, 1, 0, 0]})
```

```
In [309]: data
Out[309]:
```

	a	b
0	1	0
1	1	0
2	1	1
3	2	0
4	2	0

```
In [310]: data.value_counts()
```

```
Out[310]:
```

	a	b
1	0	2
2	0	2
1	1	1

```
dtype: int64
```

En este caso, el resultado tiene un índice que representa las diferentes filas como índice jerárquico, un tema del que hablaremos con más detalle en el capítulo 8.

5.4 Conclusión

En el siguiente capítulo, hablaremos de herramientas para leer (o cargar) y escribir conjuntos de datos con pandas. A continuación, profundizaremos más en herramientas de limpieza, disputa, análisis y visualización de datos utilizando pandas.

Carga de datos, almacenamiento y formatos de archivo

Leer datos y hacerlos accesibles (lo que se denomina carga de datos) es un primer paso necesario para utilizar la mayor parte de las herramientas de este libro. El término «análisis» se emplea también en ocasiones para describir la carga de datos de texto y su interpretación como tablas y como distintos tipos de datos. Voy a centrarme en la entrada y salida de datos mediante pandas, aunque hay herramientas en otras librerías que ayudan con la lectura y escritura de datos en diferentes formatos.

Normalmente, se puede clasificar la entrada y salida de datos en varias categorías principales: leer archivos de texto y otros formatos en disco más eficientes, cargar datos de bases de datos e interactuar con fuentes de red, como, por ejemplo, API web.

6.1 Lectura y escritura de datos en formato de texto

pandas dispone de una serie de funciones para leer datos tabulares como un objeto DataFrame. La tabla 6.1 resume algunas de ellas; `pandas.read_csv` es una de las más utilizadas en este libro. Veremos los formatos de datos binarios más tarde en este capítulo, en la sección 6.2 «Formatos de datos binarios».

Tabla 6.1. Funciones de carga de datos de texto y binarios en pandas.

Función	Descripción
<code>read_csv</code>	Carga datos delimitados de un archivo, una URL o un objeto de tipo archivo; usa la coma como delimitador predeterminado.
<code>read_fwf</code>	Lee datos en formato de columna de anchura fija (es decir, sin delimitadores).
<code>read_clipboard</code>	Variación de <code>read_csv</code> que lee datos del portapapeles; es útil para convertir tablas a partir de páginas web.
<code>read_excel</code>	Lee datos tabulares de un archivo XLS o XLSX de Excel.
<code>read_hdf</code>	Lee archivos HDF5 escritos por pandas.
<code>read_html</code>	Lee todas las tablas encontradas en el documento HTML dado.

Función	Descripción
<code>read_json</code>	Lee datos de una representación de cadena de texto, un archivo, una URL o un objeto de tipo archivo JSON (<i>JavaScript Object Notation</i> : notación de objeto JavaScript).
<code>read_feather</code>	Lee el formato de archivo binario Feather.
<code>read_orc</code>	Lee el formato de archivo binario ORC de Apache.
<code>read_parquet</code>	Lee el formato de archivo binario Parquet de Apache.
<code>read_pickle</code>	Lee un objeto almacenado por pandas empleando el formato pickle de Python.
<code>read_sas</code>	Lee un conjunto de datos SAS almacenado en uno de los formatos de almacenamiento personalizado del sistema SAS.
<code>read_spss</code>	Lee un archivo de datos creado por SPSS.
<code>read_sql</code>	Lee los resultados de una consulta SQL (utilizando SQLAlchemy).
<code>read_sql_table</code>	Lee una tabla SQL completa (utilizando SQLAlchemy); equivale a usar una consulta que lo selecciona todo en la tabla mediante <code>read_sql</code> .
<code>read_stata</code>	Lee un conjunto de datos de un formato de archivo Stata.
<code>read_xml</code>	Lee una tabla o datos de un archivo XML.

Daré un resumen general de la mecánica de estas funciones, destinadas a convertir datos de texto en un dataframe. Sus argumentos opcionales entran en varias categorías:

- **Indexación:** Puede tratar una o varias columnas como el dataframe devuelto, y decidir si obtener nombres de columnas del archivo, de los argumentos que el usuario proporciona, o de ningún argumento en absoluto.
- **Inferencia de tipos y conversión de datos:** Incluye las conversiones de valor definidas por el usuario y la lista personalizada de los marcadores de valores perdidos.
- **Análisis de fecha y hora:** Ofrece una capacidad de combinación, que incluye combinar información de fecha y hora repartida por varias columnas en una sola columna en el resultado.
- **Iteración:** Soporte para iterar por fragmentos de archivos muy grandes.
- **Problemas de datos no limpios:** Incluye saltar filas o un pie de página, comentarios u otros elementos menores, como datos numéricos con los miles separados por comas.

Debido a lo desordenados que pueden estar los datos en la realidad, parte de las funciones de carga de datos (especialmente `pandas.read_csv`) han ido acumulando

con el tiempo una larga lista de argumentos opcionales. Es normal sentirse superado por la cantidad de parámetros diferentes (`pandas.read_csv` tiene unos 50). La documentación en línea de pandas ofrece muchos ejemplos sobre cómo funcionan cada uno de ellos, de modo que si la lectura de algún archivo en particular da problemas, quizá haya un ejemplo lo bastante similar que ayude a localizar los parámetros adecuados.

Algunas de estas funciones realizan inferencia de tipos, porque los tipos de datos de las columnas no son parte del formato de datos. Esto significa que no necesariamente hay que especificar qué columnas son numéricas, enteras, booleanas o de cadena de texto. Otros formatos de datos, como HDF5, ORC y Parquet, tienen la información de los tipos de datos incrustada en el formato.

Manejar datos y otros tipos personalizados puede requerir un esfuerzo adicional.

Empecemos con un pequeño archivo de texto CSV (*Comma-Separated Values*), o de valores separados por comas.

```
In [10]: !cat examples/ex1.csv
a,b,c,d,message
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```



Aquí he utilizado el comando `cat` del shell de Unix para imprimir en pantalla el contenido del archivo sin procesar. Trabajando en Windows se puede utilizar en su lugar `type` dentro de una línea de comandos de Windows para lograr el mismo efecto.

Como está delimitado por comas, podemos usar entonces `pandas.read_csv` para leerlo en un dataframe:

```
In [11]: df = pd.read_csv("examples/ex1.csv")
```

```
In [12]: df
```

```
Out[12]:
```

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

Los archivos no siempre tienen fila de encabezado. Veamos el siguiente:

```
In [13]: !cat examples/ex2.csv
1,2,3,4,hello
5,6,7,8,world
```

```
9,10,11,12,foo
```

Para leer este archivo tenemos un par de opciones. Podemos permitir que pandas asigne nombres de columna por defecto, o bien podemos especificar nosotros los nombres:

```
In [14]: pd.read_csv("examples/ex2.csv", header=None)
Out[14]:
```

	0	1	2	3	4
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

```
In [15]: pd.read_csv("examples/ex2.csv", names=["a", "b", "c", "d",
"message"])
Out[15]:
```

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

Supongamos que queremos que la columna message sea el índice del dataframe devuelto. Podríamos o bien indicar que queremos la columna en el índice 4 o que se llame "message" utilizando el argumento `index_col`:

```
In [16]: names = ["a", "b", "c", "d", "message"]
```

```
In [17]: pd.read_csv("examples/ex2.csv", names=names,
index_col="message")
Out[17]:
```

	a	b	c	d
message				
hello	1	2	3	4
world	5	6	7	8
foo	9	10	11	12

Si queremos formar un índice jerárquico (tratado en la sección 8.1 «Indexación jerárquica») a partir de varias columnas, pasamos una lista de números o nombres de columna:

```
In [18]: !cat examples/csv_mindex.csv
```

```

key1, key2, value1, value2
one, a, 1, 2
one, b, 3, 4
one, c, 5, 6
one, d, 7, 8
two, a, 9, 10
two, b, 11, 12
two, c, 13, 14
two, d, 15, 16

```

```

In [19]: parsed = pd.read_csv("examples/csv_mindex.csv",
.....:                       index_col=["key1", "key2"])

```

```

In [20]: parsed
Out[20]:

```

		value1	value2
key1	key2		
one	a	1	2
	b	3	4
	c	5	6
	d	7	8
two	a	9	10
	b	11	12
	c	13	14
	d	15	16

En algunos casos, una tabla puede no tener un delimitador fijo y usar espacios en blanco o algún otro sistema para separar campos. Veamos un archivo de texto parecido a este:

```

In [21]: !cat examples/ex3.txt

```

	A	B	C
aaa	-0.264438	-1.026059	-0.619500
bbb	0.927272	0.302904	-0.032399
ccc	-0.264273	-0.386314	-0.217601
ddd	-0.871858	-0.348382	1.100491

Aunque se podría procesar manualmente, aquí los campos están separados por una cantidad variable de espacios en blanco. En estos casos, se puede pasar una expresión regular como delimitador para `pandas.read_csv`. Esto puede expresarse con la expresión regular `\s+`, de modo que tenemos:

```
In [22]: result = pd.read_csv("examples/ex3.txt", sep="\s+")
```

```
In [23]: result
```

```
Out[23]:
```

	A	B	C
aaa	-0.264438	-1.026059	-0.619500
bbb	0.927272	0.302904	-0.032399
ccc	-0.264273	-0.386314	-0.217601
ddd	-0.871858	-0.348382	1.100491

Como solamente había un nombre de columna menos que el número de filas de datos, en este caso especial `pandas.read_csv` infiere que la primera columna debería ser el índice del dataframe.

Las funciones de análisis de archivos tienen muchos argumentos adicionales que facilitan el manejo de la amplia variedad de formatos de archivo de excepción que ocurren (la tabla 6.2 ofrece un listado parcial). Por ejemplo, podemos saltar las filas primera, tercera y cuarta de un archivo con `skiprows`:

```
In [24]: !cat examples/ex4.csv
# oye
a,b,c,d,message
# solo quería ponerles las cosas más difíciles
# a quienes leen archivos CSV con ordenadores, ¿no?
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

```
In [25]: pd.read_csv("examples/ex4.csv", skiprows=[0, 2, 3])
```

```
Out[25]:
```

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

Manejar valores ausentes es una parte importante del proceso de lectura, a la que se suele prestar poca atención. Puede ocurrir que los datos que faltan o bien no están presentes (cadena de texto vacía) o están siendo marcados por algún valor centinela (marcador). De forma predeterminada, `pandas` usa distintos centinelas, como `NA` y `NULL`:

```
In [26]: !cat examples/ex5.csv
something,a,b,c,d,message
```

```
one,1,2,3,4,NA
two,5,6,,8,world
three,9,10,11,12,foo
```

```
In [27]: result = pd.read_csv("examples/ex5.csv")
```

```
In [28]: result
Out[28]:
```

	something	a	b	c	d	message
0	one	1	2	3.0	4	NaN
1	two	5	6	NaN	8	world
2	three	9	10	11.0	12	foo

Conviene recordar que pandas muestra los valores ausentes como NaN, de modo que tenemos dos valores nulos o faltantes en result:

```
In [29]: pd.isna(result)
Out[29]:
```

	something	a	b	c	d	message
0	False	False	False	False	False	True
1	False	False	False	True	False	False
2	False	False	False	False	False	False

La opción na_values acepta una secuencia de cadenas de texto para añadir a la lista predeterminada de cadenas de texto reconocidas como faltantes:

```
In [30]: result = pd.read_csv("examples/ex5.csv", na_values=
["NULL"])
```

```
In [31]: result
Out[31]:
```

	something	a	b	c	d	message
0	one	1	2	3.0	4	NaN
1	two	5	6	NaN	8	world
2	three	9	10	11.0	12	foo

pandas.read_csv tiene una abundante lista de representaciones de valor nulo, pero estos valores por defecto se pueden deshabilitar con la opción keep_default_na:

```
In [32]: result2 = pd.read_csv("examples/ex5.csv",
keep_default_na=False)
```

```
In [33]: result2
```

Out[33]:

	something	a	b	c	d	message
0	one	1	2	3	4	NA
1	two	5	6	8	world	
2	three	9	10	11	12	foo

In [34]: result2.isna()

Out[34]:

	something	a	b	c	d	message
0	False	False	False	False	False	False
1	False	False	False	False	False	False
2	False	False	False	False	False	False

In [35]: result3 = pd.read_csv("examples/ex5.csv",
keep_default_na=False,

.....: na_values=["NA"])

In [36]: result3

Out[36]:

	something	a	b	c	d	message
0	one	1	2	3	4	NaN
1	two	5	6	8	world	
2	three	9	10	11	12	foo

In [37]: result3.isna()

Out[37]:

	something	a	b	c	d	message
0	False	False	False	False	False	True
1	False	False	False	False	False	False
2	False	False	False	False	False	False

Es posible especificar distintos centinelas nulos para cada columna de un diccionario:

In [38]: sentinels = {"message": ["foo", "NA"], "something":
["two"]}

In [39]: pd.read_csv("examples/ex5.csv", na_values=sentinels,
.....: keep_default_na=False)

Out[39]:

	something	a	b	c	d	message
0	one	1	2	3	4	NaN
1	NaN	5	6	8	world	
2	three	9	10	11	12	foo

La tabla 6.2 lista algunas opciones utilizadas habitualmente en `pandas.read_csv`.

Tabla 6.2. Algunos argumentos de la función `pandas.read_csv`.

Argumento	Descripción
<code>path</code>	Cadena de texto que indica una ubicación en el sistema de archivos, una URL o un objeto de tipo archivo.
<code>sep</code> o <code>delimiter</code>	Secuencia de caracteres o expresión regular que se emplea para dividir campos en cada fila.
<code>header</code>	Número de fila a utilizar como nombres de columna; por defecto es 0 (primera fila), pero debería ser <code>None</code> si no hay fila de encabezado.
<code>index_col</code>	Números o nombres de columna a utilizar como índice de fila en el resultado; puede ser un solo nombre/número o una lista de ellos para un índice jerárquico.
<code>names</code>	Lista de nombres de columna para el resultado.
<code>skiprows</code>	Número de filas al comienzo del archivo que hay que ignorar o lista de números de fila (empezando por 0) que hay que saltar.
<code>na_values</code>	Secuencia de valores para reemplazar por NA. Se añaden a la lista predeterminada a menos que se pase <code>keep_default_na=False</code> .
<code>keep_default_na</code>	Si se utiliza la lista de valores NA predeterminada o no (<code>True</code> por defecto).
<code>comment</code>	Carácter o caracteres para dividir comentarios al final de las líneas.
<code>parse_dates</code>	Intenta analizar datos en <code>datetime</code> ; es <code>False</code> por defecto. Si es <code>True</code> , intentará analizar todas las columnas. En otro caso, puede especificar una lista de números o nombres de columna para analizar. Si el elemento de la lista es una tupla u otra lista, combinará varias columnas y analizará a la fecha (es decir, si la fecha u hora se divide entre dos columnas).
<code>keep_date_col</code>	Si se unen columnas para analizar la fecha, mantiene las columnas unidas; es <code>False</code> por defecto.
<code>converters</code>	Diccionario que contiene un número o nombre de columna asignado a funciones (es decir, <code>{"foo": f}</code> aplicaría la función <code>f</code> a todos los valores de la columna <code>"foo"</code>).
<code>dayfirst</code>	Cuando se analizan fechas potencialmente ambiguas, se trata como si tuvieran formato internacional (por ejemplo, <code>7/6/2012</code> -> 7 de junio de 2012); es <code>False</code> por defecto.
<code>date_parser</code>	Función que se emplea para analizar fechas.
<code>nrows</code>	Número de filas que se leen desde el principio del archivo (sin contar el encabezado).

Argumento	Descripción
iterator	Devuelve un objeto <code>TextFileReader</code> para leer el archivo por partes. Este objeto se puede utilizar también con la sentencia <code>with</code> .
chunksize	Para iteración, el tamaño de los fragmentos del archivo.
skip_footer	Número de líneas que se ignoran al final del archivo.
verbose	Imprime diversa información de análisis, como el tiempo empleado en cada etapa de la conversión del archivo e información del uso de la memoria.
encoding	Codificación de texto (por ejemplo, "utf-8" para texto codificado en UTF-8). Su valor predeterminado es "utf-8" si es <code>None</code> .
squeeze	Si los datos analizados contienen solo una columna, devuelve una serie.
thousands	Separador de miles (es decir, "," o "."); por defecto es <code>None</code> .
decimal	Separador decimal en números (es decir, "." o ","); por defecto es ".".
engine	Motor de conversión y análisis CSV; puede ser "c", "python" o "pyarrow". El valor predeterminado es "c", aunque el motor "pyarrow" más reciente puede analizar archivos mucho más rápido. El motor "python" es más lento, pero soporta funciones que los otros motores no admiten.

Leer archivos de texto por partes

Cuando se procesan archivos muy grandes o se intenta averiguar el conjunto adecuado de argumentos para procesar correctamente un archivo de gran tamaño, es conveniente leer solamente una pequeña parte del archivo o iterar a lo largo de fragmentos pequeños del archivo.

Antes de pasar a un archivo grande, haremos que la configuración de visualización de pandas sea más compacta:

```
In [40]: pd.options.display.max_rows = 10
```

Ahora tenemos:

```
In [41]: result = pd.read_csv("examples/ex6.csv")
```

```
In [42]: result
Out[42]:
```

	one	two	three	four	key
0	0.467976	-0.038649	-0.295344	-1.824726	L
1	-0.358893	1.404453	0.704965	-0.200638	B
2	-0.501840	0.659254	-0.421691	-0.057688	G
3	0.204886	1.074134	1.388361	-0.982404	R


```

4          0.354628    -0.133116    0.283763    -0.837063    Q
...          ...          ...          ...          ...          ..
9995       2.311896    -0.417070    -1.409599    -0.515821    L
9996      -0.479893    -0.650419    0.745152    -0.646038    E
9997       0.523331    0.787112    0.486066    1.093156    K
9998      -0.362559    0.598894    -1.843201    0.887292    G
9999      -0.096376    -1.012999    -0.657431    -0.573315    0
[10000 rows x 5 columns]

```

El signo de puntos suspensivos ... indica que las filas del centro del dataframe se han omitido.

Si queremos leer solo una pequeña cantidad de filas (evitando leer el archivo entero), lo especificamos con `nrows`:

```

In [43]: pd.read_csv("examples/ex6.csv", nrows=5)
Out[43]:

```

```

          one          two          three          four    key
0      0.467976    -0.038649    -0.295344    -1.824726    L
1     -0.358893     1.404453     0.704965    -0.200638    B
2     -0.501840     0.659254    -0.421691    -0.057688    G
3      0.204886     1.074134     1.388361    -0.982404    R
4      0.354628    -0.133116     0.283763    -0.837063    Q

```

Para leer un archivo por partes, especificamos un `chunksize` como número de filas:

```

In [44]: chunker = pd.read_csv("examples/ex6.csv", chunksize=1000)

In [45]: type(chunker)
Out[45]: pandas.io.parsers.readers.TextFileReader

```

El objeto `TextFileReader` devuelto por `pandas.read_csv` permite iterar a lo largo de las partes del archivo según los recuentos de valor de la columna "key", algo parecido a esto:

```

chunker = pd.read_csv("examples/ex6.csv", chunksize=1000)
tot = pd.Series([], dtype='int64')
for piece in chunker:
    tot = tot.add(piece["key"].value_counts(), fill_value=0)
tot = tot.sort_values(ascending=False)

```

Entonces tenemos:

```
In [47]: tot[:10]
Out[47]:
```

E	368.0
X	364.0
L	346.0
O	343.0
Q	340.0
M	338.0
J	337.0
F	335.0
K	334.0
H	330.0

dtype: float64

TextFileReader está también equipado con un método `get_chunk` que permite leer fragmentos de tamaño arbitrario.

Escribir datos en formato de texto

Los datos también se pueden exportar a un formato delimitado. Veamos uno de los archivos CSV leído anteriormente:

```
In [48]: data = pd.read_csv("examples/ex5.csv")
```

```
In [49]: data
Out[49]:
```

	something	a	b	c	d	message
0	one	1	2	3.0	4	NaN
1	two	5	6	NaN	8	world
2	three	9	10	11.0	12	foo

Utilizando el método `to_csv` del objeto `DataFrame`, podemos escribir los datos en un archivo separado por comas:

```
In [50]: data.to_csv("examples/out.csv")
```

```
In [51]: !cat examples/out.csv
,something,a,b,c,d,message
0,one,1,2,3.0,4,
1,two,5,6,,8,world
```

```
2, three, 9, 10, 11.0, 12, foo
```

Por supuesto, se pueden utilizar otros delimitadores (escribiendo en `sys.stdout`, de modo que imprime el resultado en texto en la consola en lugar de en un archivo):

```
In [52]: import sys

In [53]: data.to_csv(sys.stdout, sep="|")
|something|a|b|c|d|message
0|one|1|2|3.0|4|
1|two|5|6||8|world

2|three|9|10|11.0|12|foo
```

Los valores que faltan se ven como cadenas de texto vacías en el resultado. Quizá interese indicarlos con algún otro valor de centinela:

```
In [54]: data.to_csv(sys.stdout, na_rep="NULL")
, something, a, b, c, d, message
0, one, 1, 2, 3.0, 4, NULL
1, two, 5, 6, NULL, 8, world

2, three, 9, 10, 11.0, 12, foo
```

No habiendo otras opciones especificadas, se escriben tanto las etiquetas de fila como de columna, y ambas se pueden deshabilitar:

```
In [55]: data.to_csv(sys.stdout, index=False, header=False)
one, 1, 2, 3.0, 4,
two, 5, 6, , 8, world

three, 9, 10, 11.0, 12, foo
```

También se puede escribir solamente un subconjunto de las columnas, y en un orden a elección del usuario:

```
In [56]: data.to_csv(sys.stdout, index=False, columns=["a", "b",
"c"])
a, b, c
1, 2, 3.0
5, 6,
9, 10, 11.0
```

Trabajar con otros formatos delimitados

Es posible cargar la mayor parte de las formas de datos tabulares utilizando funciones como `pandas.read_csv`. Pero, en algunos casos, pueden ser necesarios

ciertos procesos manuales. No es raro recibir un archivo con una o varias líneas mal formadas que confunden a `pandas.read_csv`. Para ilustrar las herramientas básicas, veamos un pequeño archivo CSV:

```
In [57]: !cat examples/ex7.csv
"a","b","c"
"1","2","3"

"1","2","3"
```

Para cualquier archivo con un delimitador de un solo carácter, se puede emplear el módulo `csv` interno de Python. Para usarlo, basta con pasar cualquier archivo abierto u objeto de tipo archivo a `csv.reader`:

```
In [58]: import csv

In [59]: f = open("examples/ex7.csv")

In [60]: reader = csv.reader(f)
```

Iterar por el lector como un archivo produce listas de valores eliminando los que van entre comillas:

```
In [61]: for line in reader:
.....:         print(line)
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3']
In [62]: f.close()
```

A partir de ahí, es decisión propia realizar los procesos necesarios para dar a los datos la forma que necesitamos. Hagamos esto paso a paso. Primero, leemos el archivo en una lista de líneas:

```
In [63]: with open("examples/ex7.csv") as f:
.....:         lines = list(csv.reader(f))
```

Después dividimos las líneas en línea de encabezado y líneas de datos:

```
In [64]: header, values = lines[0], lines[1:]
```

Después podemos crear un diccionario de columnas de datos utilizando una comprensión de diccionario y la expresión `zip(*values)` (cuidado, porque utilizará

mucha memoria con archivos grandes), que transpone filas a columnas:

```
In [65]: data_dict = {h: v for h, v in zip(header, zip(*values))}
In [66]: data_dict
Out[66]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
```

Los archivos CSV existen en muchas clases distintas. Para definir un nuevo formato con distinto delimitador, un convenio de entrecomillado de cadenas de texto o un finalizador de línea, podríamos definir simplemente una subclase de `csv.Dialect`:

```
class my_dialect(csv.Dialect):
    lineterminator = "\n"
    delimiter = ";"
    quotechar = '"'
    quoting = csv.QUOTE_MINIMAL
    reader = csv.reader(f, dialect=my_dialect)
```

También podríamos darle a `csv.reader` parámetros individuales del dialecto CSV como palabras clave sin tener que definir una subclase:

```
reader = csv.reader(f, delimiter="|")
```

Las posibles opciones (atributos de `csv.Dialect`) y su cometido se pueden encontrar en la tabla 6.3.

Tabla 6.3. Opciones de dialect de CSV.

Argumento	Descripción
<code>delimiter</code>	Cadena de texto de un solo carácter para separar campos; su valor por defecto es <code>,</code> .
<code>lineterminator</code>	Terminador de línea para escritura; por defecto es <code>\r\n</code> . El lector lo ignora y reconoce los terminadores de línea de plataforma cruzada.
<code>quotechar</code>	Carácter de comillas para campos con caracteres especiales (como un delimitador); el valor predeterminado es <code>'</code> .
<code>quoting</code>	Convenio de entrecomillado. Las opciones disponibles son <code>csv.QUOTE_ALL</code> (entrecomillar todos los campos), <code>csv.QUOTE_MINIMAL</code> (solo campos con caracteres especiales como el delimitador), <code>csv.QUOTE_NONNUMERIC</code> y <code>csv.QUOTE_NONE</code> (sin comillas). La documentación de Python ofrece todos los detalles. Su valor por defecto es <code>csv.QUOTE_MINIMAL</code> .
<code>skipinitialspace</code>	Ignora los espacios en blanco tras cada delimitador; el valor predeterminado es <code>False</code> .

doublequote	Cómo gestionar el carácter de comillas dentro de un campo; si es True, se duplica (en la documentación en línea se puede consultar su comportamiento y resto de información).
escapechar	Cadena de texto para quitar el delimitador si quoting está fijado en csv.QUOTE_NONE; deshabilitado de forma predeterminada.



Para archivos con delimitadores de varios caracteres más complicados o fijos, no será posible usar el módulo csv. En esos casos, habrá que dividir las líneas y realizar otros arreglos utilizando el método `string` de la cadena de texto o el método de expresión regular `re.split`. Afortunadamente, `pandas.read_csv` es capaz de hacer casi todo lo necesario si se pasan las opciones correspondientes, de modo que solamente en pocos casos hará falta analizar archivos a mano.

Para escribir archivos delimitados manualmente, se puede emplear `csv.writer`. Acepta un objeto de archivo abierto en el que se puede escribir y las mismas opciones de dialecto y formato que `csv.reader`:

```
with open("mydata.csv", "w") as f:

    writer = csv.writer(f, dialect=my_dialect)
    writer.writerow(("one", "two", "three"))
    writer.writerow(("1", "2", "3"))
    writer.writerow(("4", "5", "6"))
    writer.writerow(("7", "8", "9"))
```

Datos JSON

JSON, que significa *JavaScript Object Notation* (notación de objetos de JavaScript), se ha convertido en uno de los formatos estándares para enviar datos mediante petición HTTP entre navegadores web y otras aplicaciones. Es un formato de datos mucho menos rígido que un formato de texto tabular como CSV. Aquí tenemos un ejemplo:

```
obj = """
{"name": "Wes",
 "cities_lived": ["Akron", "Nashville", "New York", "San
Francisco"],
 "pet": null,
 "siblings": [{"name": "Scott", "age": 34, "hobbies": ["guitars",
"soccer"]},
 {"name": "Katie", "age": 42, "hobbies": ["diving", "art"]}
]
"""
```

JSON es código de Python casi perfectamente válido, con la excepción de su valor nulo `null` y otras menudencias (como no permitir comas al final de las listas). Los tipos básicos son objetos (diccionarios), arrays (listas), cadenas de texto, números, valores booleanos y nulos. Todas las claves de un objeto deben ser cadenas de texto. Hay varias librerías de Python para leer y escribir datos JSON. Utilizaremos aquí `json`, ya que está integrada en la librería estándar de Python. Para convertir una cadena de texto JSON a su forma Python, empleamos `json.loads`:

```
In [68]: import json
```

```
In [69]: result = json.loads(obj)
```

```
In [70]:                                     result
Out[70]:
{'name': 'Wes',
 'cities_lived': ['Akron', 'Nashville', 'New York', 'San
Francisco'],
 'pet': None,
 'siblings': [{'name': 'Scott',
 'age': 34,
 'hobbies': ['guitars', 'soccer']}],
 'name': 'Katie', 'age': 42, 'hobbies': ['diving', 'art']}]}
```

`json.dumps`, por otro lado, convierte un objeto Python de nuevo a JSON:

```
In [71]: asjson = json.dumps(result)
```

```
In [72]: asjson
Out[72]: '{"name": "Wes", "cities_lived": ["Akron", "Nashville",
"New York", "San
Francisco"], "pet": null, "siblings": [{"name": "Scott", "age":
34, "hobbies": [
"guitars", "soccer"]}, {"name": "Katie", "age": 42, "hobbies":
["diving", "art"]}
]}'
```

La forma de convertir un objeto o lista de objetos JSON en un dataframe u otra estructura de datos para su análisis la decide el propio usuario. Resulta muy conveniente poder pasar una lista de diccionarios (que previamente eran objetos JSON) al constructor `DataFrame` y seleccionar un subconjunto de los campos de datos:

```
In [73]: siblings = pd.DataFrame(result["siblings"], columns=
["name", "age"])
```

```
In [74]: siblings
Out[74]:
```

	name	age
0	Scott	34
1	Katie	42

La función `pandas.read_json` puede convertir automáticamente conjuntos de datos JSON y colocarlos de forma específica para que formen una serie o dataframe. Por ejemplo:

```
In [75]: !cat examples/example.json
[{"a": 1, "b": 2, "c": 3},
```

```
 {"a": 4, "b": 5, "c": 6},
 {"a": 7, "b": 8, "c": 9}]
```

Las opciones predeterminadas para `pandas.read_json` suponen que cada objeto del array JSON es una fila de la tabla:

```
In [76]: data = pd.read_json("examples/example.json")
```

```
In [77]: data
Out[77]:
```

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9

El capítulo 13 incluye un ejemplo de la base de datos de alimentos del Departamento de Agricultura de los Estados Unidos para ofrecer información ampliada sobre lectura y manipulación de datos JSON (incluidos registros anidados).

Si es necesario exportar datos de pandas a JSON, una forma de hacerlo es utilizar los métodos `to_json` con series y dataframes:

```
In [78]: data.to_json(sys.stdout)
{"a":{"0":1,"1":4,"2":7},"b":{"0":2,"1":5,"2":8},"c":
{"0":3,"1":6,"2":9}}
```



```
In [79]: data.to_json(sys.stdout, orient="records")
[{"a":1,"b":2,"c":3}, {"a":4,"b":5,"c":6}, {"a":7,"b":8,"c":9}]
```

XML y HTML: raspado web

Python incluye muchas librerías para leer y escribir datos en los omnipresentes formatos HTML y XML. Algunos ejemplos son lxml, BeautifulSoup y html5lib. Aunque en general lxml es comparativamente mucho más rápido, el resto de las librerías pueden manejar mejor archivos HTML o XML mal formados.

pandas tiene una función integrada, `pandas.read_html`, que emplea todas estas librerías para analizar automáticamente tablas sacadas de archivos HTML como objetos DataFrame. Para explicar cómo funciona esto, he descargado un archivo HTML (empleado en la documentación de pandas) de la Corporación Federal de Seguro de Depósitos de los Estados Unidos que muestra quiebras de bancos¹. Primero hay que instalar algunas librerías adicionales empleadas por `read_html`:

```
conda install lxml beautifulsoup4 html5lib
```

Si no se utiliza conda, `pip install lxml` también debería funcionar.

La función `pandas.read_html` tiene varias opciones, pero por defecto busca e intenta analizar todos los datos tabulares contenidos dentro de etiquetas `<table>`. El resultado es una lista de objetos DataFrame:

```
In [80]: tables = pd.read_html("examples/fdic_failed_bank_list.html")
```

```
In [81]: len(tables)
Out[81]: 1
```

```
In [82]: failures = tables[0]
```

```
In [83]: failures.head()
Out[83]:
```

	Bank Name	City	ST	CERT	\
0	Allied Bank	Mulberry	AR	91	
1	The Woodbury Banking Company	Woodbury	GA	11297	
2	First CornerStone Bank	King of Prussia	PA	35312	
3	Trust Company Bank	Memphis	TN	9956	
4	North Milwaukee State Bank	Milwaukee	WI	20364	
	Acquiring Institution	Closing	Date	Updated	Date

0	Today's Bank	September 23,	2016	November 17,	2016
1	United Bank	August 19,	2016	November 17,	2016
2	First-Citizens Bank & Trust Company	May 6,	2016	September 6,	2016
3	The Bank of Fayette County	April 29,	2016	September 6,	2016
4	First-Citizens Bank & Trust Company	March 11,	2016	June 16,	2016

Como failures tiene muchas columnas, pandas inserta un carácter de salto de línea \.

Como aprenderemos en posteriores capítulos, desde aquí podemos proceder a realizar limpiezas y análisis varios de los datos, como por ejemplo calcular el número de quiebras de bancos al año:

```
In [84]: close_timestamps = pd.to_datetime(failures["Closing Date"])
```

```
In [85]: close_timestamps.dt.year.value_counts()
```

```
Out[85]:
```

2010	157
2009	140
2011	92
2012	51
2008	25
...	
2004	4
2001	4
2007	3
2003	3
2000	2

```
Name: Closing Date, Length: 15, dtype: int64
```

Analizar XML con lxml.objectify

XML es otro formato habitual de datos estructurados que soporta datos jerárquicos y anidados con metadatos. Este libro fue creado en realidad a partir de una serie de documentos XML de gran tamaño.

Anteriormente he hablado de la función `pandas.read_html`, que utiliza `lxml` o `Beautiful Soup` en segundo plano para analizar datos de HTML. XML y HTML son

estructuralmente similares, pero XML es más general. Aquí voy a mostrar un ejemplo de cómo utilizar lxml para analizar datos en un formato XML más general.

Durante muchos años, la MTA de Nueva York, o Autoridad Metropolitana del Transporte (*Metropolitan Transportation Authority*) estuvo publicando distintas series de datos sobre sus servicios de autobús y tren en formato XML. Vamos a ver aquí los datos de rendimiento, contenidos en varios archivos XML. Cada servicio de tren y autobús tiene un archivo distinto (como por ejemplo `Performance_MNR.xml` para el Metro-North Railroad), que incluye datos mensuales, como una serie de registros XML, con este aspecto:

```
<INDICATOR>
<INDICATOR_SEQ>373889</INDICATOR_SEQ>
<PARENT_SEQ></PARENT_SEQ>
<AGENCY_NAME>Metro-North Railroad</AGENCY_NAME>
<INDICATOR_NAME>Escalator Availability</INDICATOR_NAME>
<DESCRIPTION>Percent of the time that escalators are operational
systemwide. The availability rate is based on physical observations
performed
the morning of regular business days only. This is a new indicator
the agency
began reporting in 2009.</DESCRIPTION>
<PERIOD_YEAR>2011</PERIOD_YEAR>
<PERIOD_MONTH>12</PERIOD_MONTH>
<CATEGORY>Service Indicators</CATEGORY>
<FREQUENCY>M</FREQUENCY>
<DESIRED_CHANGE>U</DESIRED_CHANGE>
<INDICATOR_UNIT>%</INDICATOR_UNIT>
<DECIMAL_PLACES>1</DECIMAL_PLACES>
<YTD_TARGET>97.00</YTD_TARGET>
<YTD_ACTUAL></YTD_ACTUAL>
<MONTHLY_TARGET>97.00</MONTHLY_TARGET>
<MONTHLY_ACTUAL></MONTHLY_ACTUAL>
</INDICATOR>
```

Usando `lxml.objectify`, analizamos el archivo y obtenemos una referencia al nodo raíz del archivo XML con `getroot`:

```
In [86]: from lxml import objectify

In [87]: path = "datasets/mta_perf/Performance_MNR.xml"

In [88]: with open(path) as f:
.....:     parsed = objectify.parse(f)
```

```
In [89]: root = parsed.getroot()
```

root.INDICATOR devuelve un generador que produce cada elemento XML <INDICATOR>. Para cada registro, podemos llenar un diccionario de nombres de etiqueta (como YTD_ACTUAL) con valores de datos (excluyendo algunas etiquetas) ejecutando el siguiente código:

```
data = []
skip_fields = ["PARENT_SEQ", "INDICATOR_SEQ",
               "DESIRED_CHANGE", "DECIMAL_PLACES"]
for elt in root.INDICATOR:
    el_data = {}
    for child in elt.getchildren():
        if child.tag in skip_fields:
            continue
        el_data[child.tag] = child.pyval
    data.append(el_data)
```

Por último, convierte esta lista de diccionarios en un dataframe:

```
In [91]: perf = pd.DataFrame(data)
```

```
In [92]: perf.head()
```

```
Out[92]:
```

	AGENCY_NAME	INDICATOR_NAME \
0	Metro-North Railroad	On-Time Performance (West of Hudson)
1	Metro-North Railroad	On-Time Performance (West of Hudson)
2	Metro-North Railroad	On-Time Performance (West of Hudson)
3	Metro-North Railroad	On-Time Performance (West of Hudson)
4	Metro-North Railroad	On-Time Performance (West of Hudson)

	DESCRIPTION \
0	Percent of commuter trains that arrive at their destinations within 5 m...
1	Percent of commuter trains that arrive at their destinations within 5 m...
2	Percent of commuter trains that arrive at their destinations within 5 m...
3	Percent of commuter trains that arrive at their destinations

within 5 m...
 4 Percent of commuter trains that arrive at their destinations
 within 5 m...

PERIOD_YEAR	PERIOD_MONTH	CATEGORY	FREQUENCY	INDICATOR_UNIT \
0	2008	1 Service Indicators	M	%
1	2008	2 Service Indicators	M	%
2	2008	3 Service Indicators	M	%
3	2008	4 Service Indicators	M	%
4	2008	5 Service Indicators	M	%

YTD_TARGET	YTD_ACTUAL	MONTHLY_TARGET	MONTHLY_ACTUAL	
0	95.0	96.9	95.0	96.9
1	95.0	96.0	95.0	95.0
2	95.0	96.3	95.0	96.9
3	95.0	96.8	95.0	98.3
4	95.0	96.6	95.0	95.8

La función `pandas.read_xml` de `pandas` convierte este proceso en una expresión de una sola línea:

```
In [93]: perf2 = pd.read_xml(path)
```

```
In [94]: perf2.head()
```

```
Out[94]:
```

	INDICATOR_SEQ	PARENT_SEQ	AGENCY_NAME \
0	28445	NaN	Metro-North Railroad
1	28445	NaN	Metro-North Railroad
2	28445	NaN	Metro-North Railroad
3	28445	NaN	Metro-North Railroad
4	28445	NaN	Metro-North Railroad

```
INDICATOR_NAME \
```

```
0 On-Time Performance (West of Hudson)
```

1	On-Time Performance (West of Hudson)
2	On-Time Performance (West of Hudson)
3	On-Time Performance (West of Hudson)
4	On-Time Performance (West of Hudson)

DESCRIPTION \

0	Percent of commuter trains that arrive at their destinations within 5 m...
1	Percent of commuter trains that arrive at their destinations within 5 m...
2	Percent of commuter trains that arrive at their destinations within 5 m...
3	Percent of commuter trains that arrive at their destinations within 5 m...
4	Percent of commuter trains that arrive at their destinations within 5 m...

	PERIOD_YEAR	PERIOD_MONTH	CATEGORY FREQUENCY	DESIRED_CHANGE \	
0	2008	1	Service Indicators	M	U
1	2008	2	Service Indicators	M	U
2	2008	3	Service Indicators	M	U
3	2008	4	Service Indicators	M	U
4	2008	5	Service Indicators	M	U

INDICATOR_UNIT DECIMAL_PLACES YTD_TARGET YTD_ACTUAL MONTHLY_TARGET \

0	%	1	95.00	96.90	95.00
1	%	1	95.00	96.00	95.00
2	%	1	95.00	96.30	95.00
3	%	1	95.00	96.80	95.00
4	%	1	95.00	96.60	95.00

MONTHLY_ACTUAL

0	96.90
1	95.00
2	96.90
3	98.30

Para disponer de documentos XML más complejos, recomiendo consultar el docstring de `pandas.read_xml`, que describe cómo hacer selecciones y filtros para extraer una determinada tabla.

6.2 Formatos de datos binarios

Una forma sencilla de almacenar (o serializar) datos en formato binario es utilizando el módulo `pickle` interno de Python. Los objetos pandas tienen todos un método `to_pickle` que escribe los datos en disco en formato `pickle`:

```
In [95]: frame = pd.read_csv("examples/ex1.csv")
```

```
In [96]: frame
```

```
Out[96]:
```

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

```
In [97]: frame.to_pickle("examples/frame_pickle")
```

En general, los archivos `pickle` solo son legibles en Python. Se puede leer cualquier objeto con este formato almacenado en un archivo empleando el método `pickle` interno directamente, o incluso de un modo más conveniente con `pandas.read_pickle`:

```
In [98]: pd.read_pickle("examples/frame_pickle")
```

```
Out[98]:
```

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo



`pickle` solo se recomienda como formato de almacenamiento a corto plazo. El problema es que es difícil garantizar que el formato sea estable con el paso del tiempo; un objeto que está hoy en formato `pickle` puede no estarlo con una versión posterior de una librería. `pandas` ha intentado mantener la compatibilidad con versiones anteriores cuando ha sido posible, pero en algún momento del futuro puede resultar necesario «romper» el formato `pickle`.

pandas tiene soporte integrado para otros formatos de datos binarios de código abierto, como HDF5, ORC y Apache Parquet. Por ejemplo, si se instala el paquete pyarrow (conda install pyarrow), entonces se pueden leer archivos Parquet con pandas.read_parquet:

```
In [100]: fec = pd.read_parquet('datasets/fec/fec.parquet')
```

Daré algunos ejemplos del formato HDF5 en el apartado «Utilizar el formato HDF5», más adelante en este capítulo. Animo a los lectores a explorar distintos formatos para ver lo rápidos que son y lo bien que funcionan con sus análisis específicos.

Leer archivos de Microsoft Excel

pandas soporta también la lectura de datos tabulares almacenados en archivos de Excel 2003 (y versiones superiores) empleando o bien la clase pandas.ExcelFile o la función pandas.read_excel. Estas herramientas utilizan internamente los paquetes adicionales xlrd y openpyxl para leer archivos XLS antiguos y XLSX más modernos, respectivamente. Ambos deben instalarse separadamente de pandas mediante pip o conda:

```
conda install openpyxl xlrd
```

Para utilizar pandas.ExcelFile, creamos una instancia pasando una ruta a un archivo xls o xlsx:

```
In [101]: xlsx = pd.ExcelFile("examples/ex1.xlsx")
```

Este objeto puede mostrar la lista de nombres de hojas disponibles en el archivo:

```
In [102]: xlsx.sheet_names
Out[102]: ['Sheet1']
```

Los datos almacenados en una hoja se pueden leer entonces en un dataframe con parse:

```
In [103]: xlsx.parse(sheet_name="Sheet1")
Out[103]:
```

	0	a	b	c	d	message
Unnamed: 0	0	1	2	3	4	hello
1	1	5	6	7	8	world
2	2	9	10	11	12	foo

Esta tabla de Excel tiene una columna índice, de modo que podemos indicarlo con el argumento `index_col`:

```
In [104]: xlsx.parse(sheet_name="Sheet1", index_col=0)
Out[104]:
```

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

Si estamos leyendo varias hojas de un archivo, entonces es más rápido crear `pandas.ExcelFile`, pero también se le puede simplemente pasar el nombre del archivo a `pandas.read_excel`:

```
In [105]: frame = pd.read_excel("examples/ex1.xlsx",
sheet_name="Sheet1")
```

```
In [106]: frame
Out[106]:
```

Unnamed: 0	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

Para escribir datos de pandas en formato Excel, primero creamos un `ExcelWriter` y después escribimos los datos en él utilizando el método `to_excel` del objeto pandas:

```
In [107]: writer = pd.ExcelWriter("examples/ex2.xlsx")
In [108]: frame.to_excel(writer, "Sheet1")

In [109]: writer.save()
```

También se le puede pasar una ruta de archivos a `to_excel` y evitar el `ExcelWriter`:

```
In [110]: frame.to_excel("examples/ex2.xlsx")
```

Utilizar el formato HDF5

HDF5 es un formato de archivo destinado a almacenar grandes cantidades de datos de array científicos. Está disponible como librería de C, y ofrece interfaces en muchos otros lenguajes, incluyendo Java, Julia, MATLAB y Python. Sus siglas significan *Hierarchical Data Format* (formato de datos jerárquicos). Cada archivo de HDF5 puede almacenar varios conjuntos de datos y soportar metadatos. Comparado con formatos más sencillos, HDF5 admite compresión sobre la marcha con distintos modos, lo que permite almacenar datos con patrones repetidos de un modo más eficiente. HDF5 puede ser una buena opción para trabajar con conjuntos de datos que no caben en la memoria, ya que permite leer y escribir de manera eficaz pequeñas secciones de arrays mucho más grandes.

Para empezar a trabajar con HDF5 y pandas, primero hay que instalar PyTables mediante el paquete tables con conda:

```
conda install pytables
```



Hay que tener en cuenta que el paquete PyTables se llama “tables” en PyPI, de modo que si se instala con pip, será necesario ejecutar `pip install tables`.

Aunque es posible acceder directamente a archivos HDF5 utilizando las librerías PyTables o h5py, pandas ofrece una interfaz de alto nivel que simplifica el almacenamiento de objetos Series y DataFrame. La clase HDFStore funciona como un diccionario y se encarga de los detalles de bajo nivel:

```
In [113]: frame = pd.DataFrame({"a":
np.random.standard_normal(100)})

In [114]: store = pd.HDFStore("examples/mydata.h5")

In [115]: store["obj1"] = frame

In [116]: store["obj1_col"] = frame["a"]

In [117]: store
Out[117]:

<class 'pandas.io.pytables.HDFStore'>

File path: examples/mydata.h5
```

Los objetos contenidos en el archivo HDF5 pueden recuperarse después con la misma API de estilo diccionario:

```
In [118]: store["obj1"]
Out[118]:
```

	a
0	-0.204708
1	0.478943
2	-0.519439
3	-0.555730
4	1.965781
..	...
95	0.795253
96	0.118110
97	-0.748532
98	0.584970
99	0.152677

[100 rows x 1 columns]

HDFStore soporta dos esquemas de almacenamiento: “fixed” y “table” (el predeterminado es “fixed”). El último es generalmente más lento, pero soporta operaciones de consulta utilizando una sintaxis especial:

```
In [119]: store.put("obj2", frame, format="table")

In [120]: store.select("obj2", where=["index >= 10 and index <=
15"])
Out[120]:
```

	a
10	1.007189
11	-1.296221
12	0.274992
13	0.228913
14	1.352917
15	0.886429

```
In [121]: store.close()
```

La función put es una versión explícita del método store["obj2"] = frame, pero nos permite establecer otras opciones, como el formato de almacenamiento.

La función pandas.read_hdf ofrece una vía rápida para utilizar estas herramientas:

```
In [122]: frame.to_hdf("examples/mydata.h5", "obj3", format="table")

In [123]: pd.read_hdf("examples/mydata.h5", "obj3", where=["index <
5"])
Out[123]:
```

	a
0	-0.204708
1	0.478943
2	-0.519439
3	-0.555730
4	1.965781

También es posible borrar el archivo HDF5 creado haciendo lo siguiente:

```
In [124]: import os
```

```
In [125]: os.remove("examples/mydata.h5")
```



Si estamos procesando datos que se almacenan en servidores remotos, como Amazon S3 o HDFS, en este caso quizá sería más adecuado emplear otro formato binario diseñado para almacenamiento distribuido, como Apache Parquet (<http://parquet.apache.org>).

Trabajando con grandes cantidades de datos localmente, es interesante explorar PyTables y h5py para ver cómo pueden ajustarse a las necesidades particulares. Como muchos problemas de análisis de datos están relacionados con la entrada/salida (más que con la CPU), usar una herramienta como HDF5 puede acelerar masivamente las aplicaciones empleadas.



HDF5 no es una base de datos, sino que es más adecuada para conjuntos de datos de una sola escritura y varias lecturas. Aunque se pueden añadir datos a un archivo en cualquier momento, si se producen varias escrituras al mismo tiempo, el archivo puede resultar dañado.

6.3 Interactuar con API web

Muchos sitios web tienen API públicas que ofrecen feed de datos a través de JSON u algún otro formato. Hay distintas formas de acceder a estas API desde Python; un método que recomiendo es el paquete requests (<https://requests.readthedocs.io/en/latest/>), que se puede instalar con pip o conda:

```
conda install requests
```

Para encontrar los últimos 30 temas de pandas en GitHub, podemos hacer una petición GET de HTTP mediante la librería requests adicional:

```
In [126]: import requests
```

```

In [127]: url = "https://api.github.com/repos/pandas-
dev/pandas/issues"

In [128]: resp = requests.get(url)

In [129]: resp.raise_for_status()

In [130]: resp
Out[130]: <Response [200]>

```

Es una buena práctica llamar siempre a `raise_for_status` tras utilizar `requests.get` para buscar posibles errores HTTP.

El método `json` del objeto de respuesta devolverá un objeto Python que contiene los datos JSON analizados como un diccionario o una lista (según el JSON que se devuelva):

```

In [131]: data = resp.json()

In [132]: data[0]["title"]
Out[132]: 'REF: make copy keyword non-stateful'

```

Como los resultados recuperados se basan en datos en tiempo real, lo que vea cada usuario al ejecutar este código será casi seguro distinto.

Cada elemento de `data` es un diccionario que contiene todos los datos hallados en una página de temas GitHub (salvo los comentarios). Podemos pasar los datos directamente a `pandas.DataFrame` y extraer los campos que nos interesen:

```

In [133]: issues = pd.DataFrame(data, columns=["number", "title",
.....: "labels", "state"])

In [134]: issues
Out[134]:

```

	number
0	48062
1	48061
2	48060
3	48059
4	48058
..	...
25	48032
26	48030
27	48028
28	48027

```

29      48026

                                title \
0          REF: make copy keyword non-stateful
1          STYLE: upgrade flake8
2          DOC: "Creating a Python environment" in "Creating a
                development environ...
3          REGR: Avoid overflow with groupby sum
4      REGR: fix reset_index (Index.insert) regression with custom
                Index subcl...
..      ...
25      BUG: Union of multi index with EA types can lose EA dtype
26          ENH: Add rolling.prod()
27      CLN: Refactor groupby's _make_wrapper
28      ENH: Support masks in groupby prod
29      DEP: Add pip to environment.yml
                labels \
0          []
1          [{ 'id': 106935113, 'node_id': 'MDU6TGFiZWw3MDY5MzUxMTM=', 'url': 'https...
2          [{ 'id': 134699, 'node_id': 'MDU6TGFiZWw3MzQ2OTk=', 'url':
                'https://api....
3          [{ 'id': 233160, 'node_id': 'MDU6TGFiZWw3MzQ2OTk=', 'url':
                'https://api....
4          [{ 'id': 32815646, 'node_id': 'MDU6TGFiZWw3MzQ2OTk=', 'url': 'https:...
..      ...
25      [{ 'id': 76811, 'node_id': 'MDU6TGFiZWw3NjgxMQ==', 'url':
                'https://api.g...
26      [{ 'id': 76812, 'node_id': 'MDU6TGFiZWw3NjgxMQ==', 'url':
                'https://api.g...
27      [{ 'id': 233160, 'node_id': 'MDU6TGFiZWw3MzQ2OTk=', 'url':
                'https://api....
28      [{ 'id': 233160, 'node_id': 'MDU6TGFiZWw3MzQ2OTk=', 'url':
                'https://api....
29      [{ 'id': 76811, 'node_id': 'MDU6TGFiZWw3NjgxMQ==', 'url':
                'https://api.g...

state
0      open
1      open
2      open
3      open
4      open
..      ...
25      open

```

```
26      open
27      open
28      open
29      open
[30 rows x 4 columns]
```

Con un poco de esfuerzo se pueden crear ciertas interfaces de alto nivel para API web habituales, que devuelven objetos DataFrame para un análisis más conveniente.

6.4 Interactuar con bases de datos

En un entorno de empresa, muchos datos pueden no almacenarse en archivos Excel o de texto. Las bases de datos relacionales basadas en SQL (como SQL Server, PostgreSQL y MySQL) son de uso general, aunque también hay muchas bases de datos alternativas que están adquiriendo gran popularidad. La elección de base de datos depende normalmente de las necesidades de rendimiento, integridad de datos y escalabilidad de una aplicación.

pandas incluye varias funciones para simplificar la carga de los resultados de una consulta SQL en un dataframe. Como ejemplo, crearé una base de datos SQLite3 utilizando el controlador sqlite3 interno de Python:

```
In [135]: import sqlite3

In [136]:
.....: query = """
.....: CREATE TABLE test
.....: (a VARCHAR(20), b VARCHAR(20),
.....: c REAL, d INTEGER
.....: );"""

In [137]: con = sqlite3.connect("mydata.sqlite")

In [138]: con.execute(query)
Out[138]: <sqlite3.Cursor at 0x7fd73b69c0>

In [139]: con.commit()
```

Después insertamos varias filas de datos:

```
In [140]: data = [("Atlanta", "Georgia", 1.25, 6),
.....: ("Tallahassee", "Florida", 2.6, 3),
```

```

.....:      ("Sacramento", "California", 1.7, 5)]

In [141]: stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"

In [142]: con.executemany(stmt, data)
Out[142]: <sqlite3.Cursor at 0x7fd73a00c0>

In [143]: con.commit()

```

La mayoría de los controladores SQL de Python devuelven una lista de tuplas al seleccionar datos de una tabla:

```

In [144]: cursor = con.execute("SELECT * FROM test")

In [145]: rows = cursor.fetchall()

In [146]: rows
Out[146]:

[('Atlanta', 'Georgia', 1.25, 6),
 ('Tallahassee', 'Florida', 2.6, 3),
 ('Sacramento', 'California', 1.7, 5)]

```

Se puede pasar la lista de tuplas al constructor DataFrame, pero también se necesitan los nombres de columnas, contenidos en el atributo description del cursor. Hay que tener en cuenta que para SQLite3, el atributo description del cursor solamente ofrece nombres de columnas (los otros campos, que son parte de la especificación Database API de Python, son None), pero con otros controladores de bases de datos se dispone de más información sobre las columnas:

```

In [147]: cursor.description
Out[147]:

(('a', None, None, None, None, None, None),
 ('b', None, None, None, None, None, None),
 ('c', None, None, None, None, None, None),
 ('d', None, None, None, None, None, None))

In [148]: pd.DataFrame(rows, columns=[x[0] for x in
Out[148]:

```

	a	b	c	d
0	Atlanta	Georgia	1.25	6
1	Tallahassee	Florida	2.60	3
2	Sacramento	California	1.70	5

Quizá sea preferible no repetir este procesado cada vez que se consulta la base de datos. El proyecto SQLAlchemy (<http://www.sqlalchemy.org/>) es un conocido kit de herramientas SQL de Python que resume muchas de las diferencias normales entre bases de datos SQL. pandas tiene una función `read_sql` que permite leer datos fácilmente desde una conexión SQLAlchemy general. Se puede instalar SQLAlchemy con conda del siguiente modo:

```
conda install sqlalchemy
```

Ahora conectaremos con la misma base de datos SQLite con SQLAlchemy y leeremos datos desde la tabla creada anteriormente:

```
In [149]: import sqlalchemy as sqla
```

```
In [150]: db = sqla.create_engine("sqlite:///mydata.sqlite")
```

```
In [151]: pd.read_sql("SELECT * FROM test", db)
```

```
Out[151]:
```

	a	b	c	d
0	Atlanta	Georgia	1.25	6
1	Tallahassee	Florida	2.60	3
2	Sacramento	California	1.70	5

6.5 Conclusión

Obtener acceso a los datos suele ser el primer paso en el proceso de análisis de datos. Hemos visto en este capítulo distintas herramientas útiles para tal objetivo. En los siguientes capítulos profundizaremos en la disputa y visualización de datos, en el análisis de series temporales y en otros temas.

¹ Para consultar la lista completa véase <https://www.fdic.gov/bank/individual/failed/banklist.html>.

Limpieza y preparación de los datos

Mientras se realiza análisis y modelado de datos, se emplea una considerable cantidad de tiempo en la preparación de los mismos: carga, limpieza, transformación y reordenación. A menudo estas tareas le ocupan al analista más del 80 % de su tiempo. En ocasiones, los archivos o bases de datos en los que estos se almacenan no están en el formato adecuado para una determinada tarea. Muchos investigadores eligen procesar los datos a medida con un lenguaje de programación de uso general, como Python, Perl, R o Java, o bien con herramientas de proceso de texto de Unix, como sed o awk. Por suerte, pandas, junto con las funciones internas del lenguaje Python, ofrece un juego de herramientas de alto nivel, flexible y rápido para manipular datos del modo adecuado.

Si algún lector identifica un tipo de manipulación de datos que no esté incluido en este libro o en ningún punto de la librería pandas, por favor no duden en compartir el caso en una de las listas de correo de Python o en el sitio GitHub de pandas. Es una realidad que las necesidades de aplicaciones reales han impulsado buena parte del diseño y la implementación de pandas.

En este capítulo hablaré de las herramientas para datos ausentes o faltantes, datos duplicados, manipulación de cadenas de texto y otras transformaciones de datos analíticas. En el siguiente capítulo me centraré en la combinación y reordenación de conjuntos de datos de distintas formas.

7.1 Gestión de los datos que faltan

En muchas aplicaciones de análisis de datos suele haber datos ausentes, faltantes o perdidos. Uno de los objetivos de pandas es que el trabajo con este tipo de datos sea lo más sencillo posible. Por ejemplo, todas las

estadísticas descriptivas sobre objetos pandas dejan fuera a los datos ausentes de forma predeterminada.

La forma en la que se representan los datos ausentes en objetos pandas es ciertamente imperfecta, pero es suficiente para la mayoría de los usos en el mundo real. Para datos de tipo `float64` en la propiedad `dtype`, pandas emplea el valor de punto flotante `NaN` (Not a Number: no es un número) para representar datos que faltan.

Llamamos a este valor centinela: cuando está presente, indica un valor faltante (o nulo):

```
In [14]: float_data = pd.Series([1.2, -3.5, np.nan, 0])
```

```
In [15]: float_data
```

```
Out[15]:
```

0	1.2
1	-3.5
2	NaN
3	0.0
dtype:	float64

El método `isna` nos da una serie booleana con `True`, en la que los valores son nulos:

```
In [16]: float_data.isna()
```

```
Out[16]:
```

0	False
1	False
2	True
3	False
dtype:	bool

En pandas hemos adoptado un convenio empleado en el lenguaje de programación R refiriéndonos a los datos ausentes como `NA`, que significa Not Available (no disponible). En aplicaciones de estadística, los datos `NA`

pueden ser o bien datos que no existen o que existen pero no se han observado (debido a problemas con la recogida de datos, por ejemplo). Al limpiar datos para su análisis, suele ser importante analizar también los datos que no están, para identificar problemas en la recogida de datos o posibles desviaciones en los datos producidas por datos faltantes.

El valor None interno de Python se trata también como NA:

```
In [17]: string_data = pd.Series(["aardvark", np.nan, None,
    "avocado"])
```

```
In [18]: string_data
Out[18]:
```

0	aardvark
1	NaN
2	None
3	avocado
dtype:	object

```
In [19]: string_data.isna()
Out[19]:
```

0	False
1	True
2	True
3	False
dtype:	bool

```
In [20]: float_data = pd.Series([1, 2, None],
    dtype='float64')
```

```
In [21]: float_data
Out[21]:
```

0	1.0
1	2.0
2	NaN
dtype:	float64

```
In [22]: float_data.isna()
Out[22]:
```

```
0          False
1          False
2           True
dtype: bool
```

El proyecto pandas ha intentado que el trabajo con datos ausentes sea consistente en los distintos tipos de datos. Funciones como `pandas.isna` aíslan muchos de los molestos detalles. En la tabla 7.1 podemos ver una lista de algunas funciones relacionadas con la manipulación de datos ausentes.

Tabla 7.1. Métodos de objeto para gestionar valores nulos.

Método	Descripción
<code>dropna</code>	Filtra etiquetas de eje basándose en si los valores de cada etiqueta tienen datos ausentes, con diversos umbrales para la cantidad de datos nulos que soportar.
<code>fillna</code>	Rellena datos faltantes con algún valor o utilizando un método de interpolación como "ffill" o "bfill".
<code>isna</code>	Devuelve valores booleanos indicando qué valores están ausentes o son nulos.
<code>notna</code>	Negación de <code>isna</code> , devuelve <code>True</code> para valores que no son nulos y <code>False</code> para los que lo son.

Filtrado de datos que faltan

Hay varias formas de filtrar datos ausentes. Aunque siempre se disponga de la opción de hacerlo a mano utilizando `pandas.isna` e `indexado booleano`, `dropna` puede ser útil. En un objeto `Series`, devuelve la serie solo con los valores de datos e índice no nulos:

```
In [23]: data = pd.Series([1, np.nan, 3.5, np.nan, 7])
```

```
In [24]: data.dropna()
```

```
Out[24]:
```

0	1.0
2	3.5
4	7.0
dtype:	float64

Esto es lo mismo que hacer lo siguiente:

```
In [25]: data[data.notna()]
```

```
Out[25]:
```

0	1.0
2	3.5
4	7.0
dtype:	float64

Con objetos DataFrame, hay distintas maneras de eliminar los datos que faltan. Quizá interese eliminar las filas o columnas que son todas nulas, o solamente las filas o columnas que contengan algún valor nulo. `dropna` elimina por defecto cualquier fila que contiene un valor faltante:

```
In [26]: data = pd.DataFrame([[1., 6.5, 3.], [1., np.nan, np.nan],  
.....: [np.nan, np.nan, np.nan], [np.nan, 6.5, 3.]])
```

```
In [27]: data
```

```
Out[27]:
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

```
In [28]: data.dropna()
```

```
Out[28]:
```

	0	1	2
0	1.0	6.5	3.0

Pasar `how="all"` quitará solamente las filas que sean todas nulas:

```
In [29]: data.dropna(how="all")
Out[29]:
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
3	NaN	6.5	3.0

Conviene recordar que estas funciones devuelven objetos nuevos de forma predeterminada y no modifican el contenido del objeto original.

Para eliminar columnas del mismo modo, pasamos `axis="columns"`:

```
In [30]: data[4] = np.nan
```

```
In [31]: data
Out[31]:
```

	0	1	2	4
0	1.0	6.5	3.0	NaN
1	1.0	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN
3	NaN	6.5	3.0	NaN

```
In [32]: data.dropna(axis="columns", how="all")
Out[32]:
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

Supongamos que queremos mantener solo filas que contengan como máximo un cierto número de observaciones faltantes. Se puede indicar esto con el argumento thresh:

```
In [33]: df = pd.DataFrame(np.random.standard_normal((7, 3)))
```

```
In [34]: df.iloc[:4, 1] = np.nan
```

```
In [35]: df.iloc[:2, 2] = np.nan
```

```
In [36]: df
```

```
Out[36]:
```

	0	1	2
0	-0.204708	NaN	NaN
1	-0.555730	NaN	NaN
2	0.092908	NaN	0.769023
3	1.246435	NaN	-1.296221
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

```
In [37]: df.dropna()
```

```
Out[37]:
```

	0	1	2
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

```
In [38]: df.dropna(thresh=2)
```

```
Out[38]:
```

	0	1	2
2	0.092908	NaN	0.769023
3	1.246435	NaN	-1.296221
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

Rellenado de datos ausentes

En lugar de filtrar datos ausentes (y posiblemente arrastrar con ellos otros datos), quizá sea más conveniente rellenar los “huecos” de distintas maneras. Para la mayoría de los casos se debe emplear el método `fillna`. Llamar a `fillna` con una constante reemplaza los valores ausentes por dicho valor:

```
In [39]: df.fillna(0)
```

```
Out[39]:
```

	0	1	2
0	-0.204708	0.000000	0.000000
1	-0.555730	0.000000	0.000000
2	0.092908	0.000000	0.769023
3	1.246435	0.000000	-1.296221
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

Llamando a `fillna` con un diccionario se puede utilizar un valor de relleno distinto para cada columna:

```
In [40]: df.fillna({1: 0.5, 2: 0})
```

```
Out[40]:
```

	0	1	2
0	-0.204708	0.500000	0.000000
1	-0.555730	0.500000	0.000000
2	0.092908	0.500000	0.769023
3	1.246435	0.500000	-1.296221
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

Los mismos métodos de interpolación disponibles para reindexar (véase la tabla 5.3) pueden usarse con `fillna`:

```
In [41]: df = pd.DataFrame(np.random.standard_normal((6, 3)))
```

```
In [42]: df.iloc[2:, 1] = np.nan
```

```
In [43]: df.iloc[4:, 2] = np.nan
```

```
In [44]: df
```

```
Out[44]:
```

	0	1	2
0	0.476985	3.248944	-1.021228
1	-0.577087	0.124121	0.302614
2	0.523772	NaN	1.343810
3	-0.713544	NaN	-2.370232
4	-1.860761	NaN	NaN
5	-1.265934	NaN	NaN

```
In [45]: df.fillna(method="ffill")
```

```
Out[45]:
```

	0	1	2
0	0.476985	3.248944	-1.021228
1	-0.577087	0.124121	0.302614
2	0.523772	0.124121	1.343810
3	-0.713544	0.124121	-2.370232
4	-1.860761	0.124121	-2.370232
5	-1.265934	0.124121	-2.370232

```
In [46]: df.fillna(method="ffill", limit=2)
```

```
Out[46]:
```

	0	1	2
0	0.476985	3.248944	-1.021228
1	-0.577087	0.124121	0.302614
2	0.523772	0.124121	1.343810
3	-0.713544	0.124121	-2.370232
4	-1.860761	NaN	-2.370232
5	-1.265934	NaN	-2.370232

Con `fillna` se pueden hacer muchas otras cosas, como la imputación de datos sencilla utilizando las estadísticas de mediana o media:

```
In [47]: data = pd.Series([1., np.nan, 3.5, np.nan, 7])
```

```
In [48]: data.fillna(data.mean())
```

```
Out[48]:
```

```
0          1.000000
1          3.833333
2          3.500000
3          3.833333
4          7.000000
dtype: float64
```

Véase en la tabla 7.2 una referencia sobre los argumentos de la función `fillna`.

Tabla 7.2. Argumentos de la función `fillna`.

Argumento	Descripción
<code>value</code>	Valor escalar u objeto de tipo diccionario que se utiliza para rellenar valores faltantes.
<code>method</code>	Método de interpolación: puede ser <code>"bfill"</code> (relleno hacia atrás) o <code>"ffill"</code> (relleno hacia delante); el valor predeterminado es <code>None</code> .
<code>axis</code>	Eje que rellenar (<code>"index"</code> o <code>"columns"</code>); el valor predeterminado es <code>axis="index"</code> .
<code>limit</code>	Para relleno hacia delante o hacia atrás, máximo número de períodos consecutivos que rellenar.

7.2 Transformación de datos

Hasta ahora en este capítulo nos hemos centrado sobre todo en la gestión de datos ausentes. Pero hay otras transformaciones, como el filtrado y la limpieza, que también son operaciones importantes con datos.

Eliminación de duplicados

En un objeto DataFrame se pueden encontrar filas duplicadas por distintas razones. Aquí tenemos un ejemplo:

```
In [49]: data = pd.DataFrame({"k1": ["one", "two"] * 3 +  
    ....: "k2": [1, 1, 2, 3, 3, 4, 4]})
```

```
In [50]: data  
Out[50]:
```

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4
6	two	4

El método `uplicated` de DataFrame devuelve una serie booleana indicando si cada fila es un duplicado (sus valores de columna son exactamente iguales a los de una fila anterior) o no:

```
In [51]: data.duplicated()  
Out[51]:
```

0	False
1	False
2	False
3	False
4	False
5	False
6	True
dtype:	bool

Algo parecido hace `drop_duplicates`, que devuelve un dataframe con filas en las que el array `duplicate` es `False` al ser filtrado:

```
In [52]: data.drop_duplicates()  
Out[52]:
```

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4

Ambos métodos tienen en cuenta todas las columnas de forma predeterminada; como alternativa se puede especificar cualquier subconjunto de ellas para detectar duplicados. Supongamos que tenemos una columna adicional de valores y queremos filtrar los duplicados basándonos solamente en la columna “k1”:

```
In [53]: data["v1"] = range(7)
```

```
In [54]: data  
Out[54]:
```

	k1	k2	v1
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
5	two	4	5
6	two	4	6

```
In [55]: data.drop_duplicates(subset=["k1"])  
Out[55]:
```

k1	k2	v1
----	----	----

0	one	1	0
1	two	1	1

`data.duplicated` y `data.drop_duplicates` conservan por defecto la primera combinación de valor observada. Pasar `keep="last"` devolverá la última:

```

In [56]: data.drop_duplicates(["k1", "k2"], keep="last")
Out[56]:
  
```

	k1	k2	v1
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
6	two	4	6

Transformación de datos mediante una función o una asignación

Para muchos conjuntos de datos quizá nos interese más realizar transformaciones basadas en los valores de un array, una serie o una columna en un dataframe. Veamos los siguientes datos hipotéticos recogidos sobre distintos tipos de carne:

```

In [57]: data = pd.DataFrame({"food": ["bacon", "pulled
pork", "bacon",
....: "pastrami", "corned beef", "bacon",
....: "pastrami", "honey ham", "nova lox"],
....: "ounces": [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
  
```

```

In [58]: data
Out[58]:
  
```

	food	ounces
0	bacon	4.0
1	pulled pork	3.0
2	bacon	12.0
3	pastrami	6.0

4	corned beef	7.5
5	bacon	8.0
6	pastrami	3.0
7	honey ham	5.0
8	nova	6.0

Supongamos que queremos añadir una columna indicando el tipo de animal del que procede cada alimento. Asignemos cada tipo de carne al tipo de animal:

```
meat_to_animal = {
    "bacon": "pig",
    "pulled pork": "pig",
    "pastrami": "cow",
    "corned beef": "cow",
    "honey ham": "pig",
    "nova lox": "salmon"
}
```

El método `map` de un objeto `Series` (del que también hemos hablado en la sección «Aplicación y asignación de funciones» en el capítulo 5) acepta una función o un objeto de tipo diccionario que contiene un mapeado para hacer la transformación de los valores:

```
In [60]: data["animal"] = data["food"].map(meat_to_animal)
```

```
In [61]: data
```

```
Out[61]:
```

	food	ounces	animal
0	bacon	4.0	pig
1	pulled pork	3.0	pig
2	bacon	12.0	pig
3	pastrami	6.0	cow
4	corned beef	7.5	cow
5	bacon	8.0	pig
6	pastrami	3.0	cow

7	honey ham	5.0	pig
8	nova lox	6.0	salmon

También podríamos haber pasado una función que haga todo el trabajo:

```
In [62]: def get_animal(x):
.....: return meat_to_animal[x]
```

```
In [63]: data["food"].map(get_animal)
```

```
Out[63]:
```

	pig
0	
1	pig
2	pig
3	cow
4	cow
5	pig
6	cow
7	pig
8	salmon

```
Name: food, dtype: object
```

Utilizar `map` es una forma conveniente de realizar transformaciones elemento a elemento y otras operaciones de datos relacionadas con su limpieza.

Reemplazar valores

Rellenar datos ausentes con el método `fillna` es un caso especial del reemplazo de valores más general. Como ya hemos visto, `map` se puede utilizar para modificar un subconjunto de valores de un objeto, pero `replace` ofrece una forma más sencilla y flexible de hacerlo. Veamos esta serie:

```
In [64]: data = pd.Series([1., -999., 2., -999., -1000.,
3.])
```



```
In [65]: data
Out[65]:
```

```
0          1.0
1        -999.0
2          2.0
3        -999.0
4       -1000.0
5          3.0
dtype: float64
```

Los valores -999 podrían ser valores centinela para datos ausentes. Para reemplazarlos por valores NA que pandas comprenda, podemos utilizar `replace`, produciendo una nueva serie:

```
In [66]: data.replace(-999, np.nan)
Out[66]:
```

```
0          1.0
1          NaN
2          2.0
3          NaN
4       -1000.0
5          3.0
dtype: float64
```

Si queremos reemplazar varios valores al mismo tiempo, lo que hacemos es pasar una lista y después el valor sustituto:

```
In [67]: data.replace([-999, -1000], np.nan)
Out[67]:
```

```
0          1.0
1          NaN
2          2.0
3          NaN
4          NaN
```

```
5
dtype: float64 3.0
```

Para utilizar un sustituto distinto para cada valor, pasamos una lista de sustitutos:

```
In [68]: data.replace([-999, -1000], [np.nan, 0])
Out[68]:
```

0	1.0
1	NaN
2	2.0
3	NaN
4	0.0
5	3.0
dtype:	float64

El argumento pasado puede ser también un diccionario:

```
In [69]: data.replace({'-999': np.nan, '-1000': 0})
Out[69]:
```

0	1.0
1	NaN
2	2.0
3	NaN
4	0.0
5	3.0
dtype:	float64



El método `data.replace` es distinto de `data.str.replace`, que reemplaza cadenas de texto elemento a elemento. Veremos estos métodos de cadena de texto con objetos Series más adelante en el capítulo.

Renombrar índices de eje

Igual que los valores de una serie, las etiquetas de eje se pueden transformar de una forma parecida mediante una función o asignación de algún tipo, para producir nuevos objetos con etiqueta diferente. También se pueden modificar los ejes en el momento sin crear una nueva estructura de datos. Aquí tenemos un ejemplo sencillo:

```
In [70]: data = pd.DataFrame(np.arange(12).reshape((3, 4)),
.....:      index=["Ohio", "Colorado", "New York"],
.....:      columns=["one", "two", "three", "four"])
```

Igual que en una serie, los índices de eje tienen un método map:

```
In [71]: def transform(x):
.....:     return x[:4].upper()

In [72]: data.index.map(transform)
Out[72]: Index(['OHIO', 'COLO', 'NEW'], dtype='object')
```

Podemos asignar el atributo index, modificando el dataframe en el acto:

```
In [73]: data.index = data.index.map(transform)

In [74]: data
Out[74]:
```

	one	two	three	four
OHIO	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

Si queremos crear una versión transformada de un conjunto de datos sin modificar el original, un método útil es rename:

```
In [75]: data.rename(index=str.title, columns=str.upper)
Out[75]:
```

	ONE	TWO	THREE	FOUR
Ohio	0	1	2	3

Colo	4	5	6	7
New	8	9	10	11

Vale la pena mencionar que `rename` se puede emplear junto con un objeto de tipo diccionario, proporcionando así nuevos valores para un subconjunto de las etiquetas de eje:

```
In [76]: data.rename(index={"OHIO": "INDIANA"},
.....: columns={"three": "peekaboo"})
Out[76]:
```

	one	two	peekaboo	four
INDIANA	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

`rename` evita la tarea de copiar el dataframe a mano y asignar nuevos valores a sus atributos `index` y `columns`.

Discretización

Los datos continuos se suelen discretizar o, lo que es lo mismo, separar en «contenedores» para su análisis. Imaginemos que tenemos datos sobre un grupo de personas de un estudio, y queremos agruparlos en sendos contenedores por edad:

```
In [77]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41,
32]
```

Dividámoslos en otros contenedores de 18 a 25 años, 26 a 35 años, 36 a 60 años y finalmente 61 años o más. Para ello tenemos que usar `pandas.cut`:

```
In [78]: bins = [18, 25, 35, 60, 100]
```

```
In [79]: age_categories = pd.cut(ages, bins)
```

```
In [80]: age_categories
```