

Actualización 2023

O'REILLY® ANAYA
MULTIMEDIA

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 ocn padas, 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 Massachussets 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 *Tupaiidae*. 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 Augspurser, 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 Augspurser, 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 Pilssworth, 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 Sargent, 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

Compreensiones 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, quizás 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 «pegamiento» 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, quizás 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 comparten 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
```

Cuando 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 quizás 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 manejarlos 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 quizás 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=0a77b52fefef52ab83e3c35dff8de121e4bb443a63f2d...
[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=0a77b52fefef52ab83e3c35dff8de121e4...
or http://127.0.0.1:8888/?token=0a77b52fefef52ab83e3c35dff8de121e4...
```

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=0a77b52fefef52ab83e3c35dff8de121e4bb443a63f2d3055>. 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.

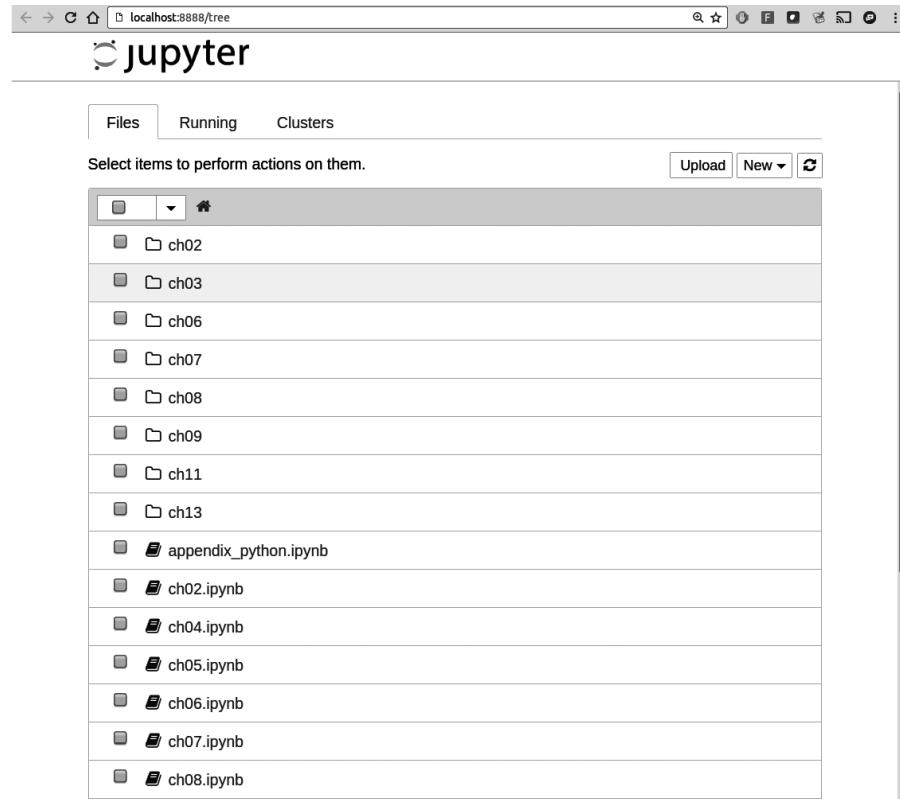


Figura 2.1. Página de inicio de Jupyter notebook.

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.

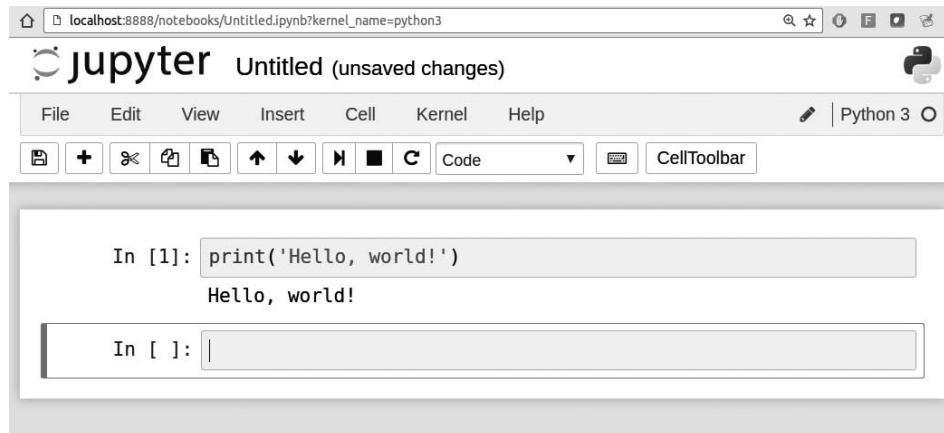


Figura 2.2. Vista de un nuevo notebook de Jupyter.

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.

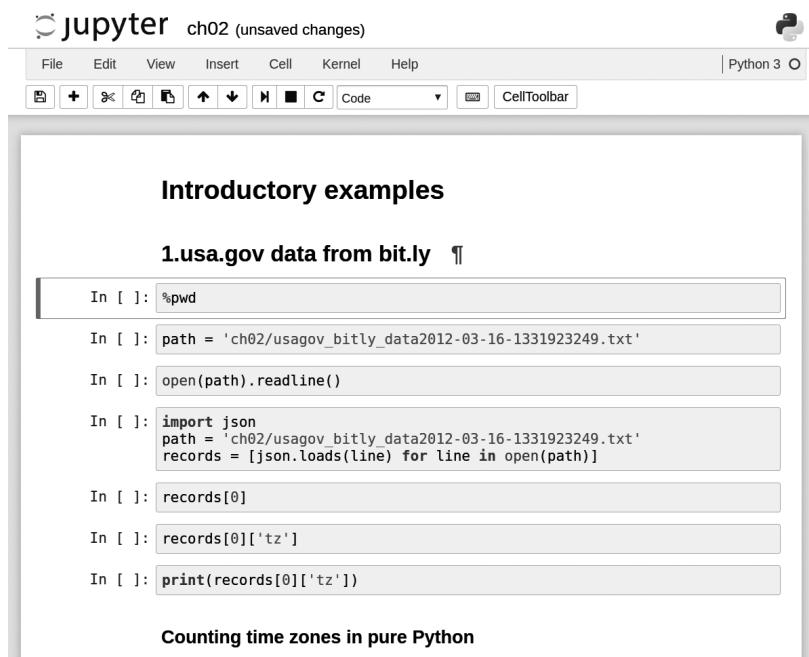


Figura 2.3. Vista de ejemplo de Jupyter de un notebook existente.

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>
```

append()	count()	insert()	reverse()
clear()	extend()	pop()	sort()
copy()	index()	remove()	

Lo mismo aplica a los módulos:

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

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

date	MAXYEAR	timedelta
datetime	MINYEAR	timezone
datetime_CAPI	time	tzinfo



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).

La captura de pantalla muestra una terminal de Jupyter Notebook. En la primera línea, se ve el código `In [12]: def func_with_keywords(abra=1, abbra=2, abbbra=3): return abra, abbra, abbbra`. La segunda línea muestra la ejecución `In []: func_with_keywords(ab|)`, donde el cursor está en el carácter 'b' de 'abra'. Una barra deslizante aparece debajo de la línea de入力, mostrando las palabras clave `abra=`, `abbra=`, `abbbra=` y `abs`.

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 (¡activelo!). 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 quizás 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]
```

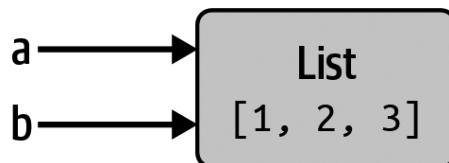


Figura 2.5. Dos referencias al mismo objeto.

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 sobreescribirá 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
--> 2 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>  
  
capitalize()    index()        isspace()       removesuffix()  startswith()  
casefold()      isprintable()   istitle()       replace()       strip()  
center()        isalnum()       isupper()       rfind()        swapcase()  
count()         isalpha()       join()         rindex()       title()  
encode()        isascii()      ljust()        rjust()        translate()  
endswith()      isdecimal()    lower()        rpartition()  
expandtabs()    isdigit()      lstrip()       rsplit()  
find()          isidentifier() maketrans()    rstrip()  
format()        islower()      partition()    split()  
format_map()    isnumeric()    removeprefix() splitlines()
```

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	Multiplica 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
a is b	True si a y b hacen referencia al mismo objeto Python
a is not b	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 casi 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"
--> 2 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\no\special\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'espac\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\x00\x00n\x00a\x00\f1\x00o\x00\x00'
```

```
In [94]: val.encode("utf-16le")
Out[94]: b'e\x0005\x00n\x00a\x00\xf1\x00o\x001\x00'
```

Es más habitual encontrar objetos bytes cuando se trabaja con archivos, donde quizás 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(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.

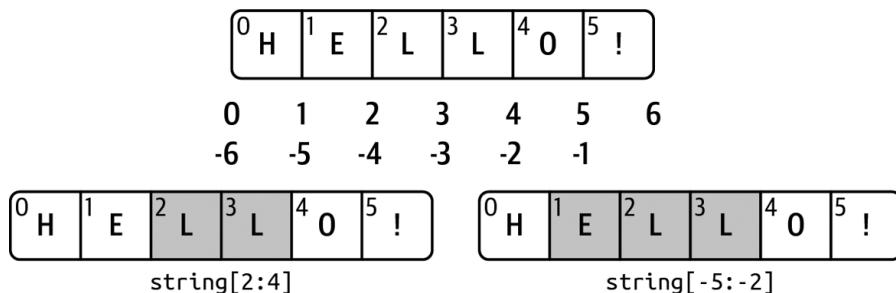


Figura 3.1. Ilustración de los convenios de corte de Python.

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>	<code>a b</code>	Todos los elementos únicos de a y b
<code>a.update(b)</code>	<code>a = b</code>	Fija el contenido de a para que sea la unión de los elementos de a y b
<code>a.intersection(b)</code>	<code>a & b</code>	Todos los elementos tanto de a como de b
<code>a.intersection_update(b)</code>	<code>a &= b</code>	Fija el contenido de a para que sea la intersección de los elementos de a y b
<code>a.difference(b)</code>	<code>a - b</code>	Los elementos de a que no están en b
<code>a.difference_update(b)</code>	<code>a -= b</code>	Fija en a los elementos de a que no están en b
<code>a.symmetric_difference(b)</code>	<code>a ^ b</code>	Todos los elementos de a o b pero no de los dos
<code>a.symmetric_difference_update(b)</code>	<code>a ^= b</code>	Fija a para que contenga los elementos de a o b pero no de ambos
<code>a.issubset(b)</code>	<code><=</code>	True si los elementos de a están todos contenidos en b
<code>a.issuperset(b)</code>	<code>>=</code>	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          for index, (a, b) in enumerate(zip(seq1,
[153]:      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`).

Comprendiones 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 desempaquetá 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 k 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 k 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")
--> 1 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))
--> 1 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-230-6209ddecfd2b5> 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>()
           1     def throws_an_exception():
           2         raise AssertionError("This is a test")
```

```

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 procesado 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>	Vacia 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
    321          # cuenta el búfer)
    321      data = self.buffer + input
->   322      (result,           consumed)      =
    322          self._buffer_decode(data, self.errors,
    323          final
)
    323          # mantiene sin decodificar la entrada
    323          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 `put 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.])
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
array	Convierte datos de entrada (lista, tupla, array u otro tipo de secuencia) en un ndarray o bien deduciendo un tipo de datos o especificándolo de forma explícita; copia los datos de entrada por omisión.
asarray	Convierte la entrada en ndarray, pero no copia si la entrada ya es un ndarray.
arange	Igual que la función range interna, pero devuelve un ndarray en lugar de una lista.
ones, ones_like	Produce un array de todo unos con la forma y el tipo de datos dados; ones_like toma otro array y produce un array ones con la misma forma y tipo de datos.
zeros, zeros_like	Igual que ones y ones_like, pero produciendo arrays de ceros.
empty, empty_like	Crea nuevos arrays asignando nueva memoria, pero no los llena con valores como ones y zeros.
full, full_like	Produce un array con la forma y el tipo de datos dados y con todos los valores fijados en el «valor de relleno» indicado; full_like toma otro array y produce un array llenado con la misma forma y tipo de datos.
eye, identity	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 o f</code>	Punto flotante estándar de precisión sencilla; compatible con <code>float</code> de C.
<code>float64</code>	<code>f8 o d</code>	Punto flotante estándar de precisión doble; compatible con <code>double</code> de C y el objeto <code>float</code> de Python.

<code>float128</code>	<code>f16 o g</code>	Punto flotante de precisión extendida.
<code>complex64,</code> <code>complex128,</code> <code>complex256</code>	<code>c8, c16,</code> <code>c32</code>	Números complejos representados por dos floats de 32, 64 y 120, respectivamente.
<code>bool</code>	<code>?</code>	Tipo booleano que almacena valores <code>True</code> y <code>False</code> .
<code>object</code>	<code>o</code>	Tipo de objeto Python; un valor puede ser cualquier objeto Python.
<code>string_</code>	<code>s</code>	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 " <code>s10</code> ".
<code>unicode_</code>	<code>u</code>	Tipo Unicode de longitud fija (el número de bytes es específico de la plataforma); la misma semántica de especificación que <code>string_</code> (por ejemplo, " <code>u10</code> ").



Existen tipos enteros con y sin signo, y quizás 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	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]])
```

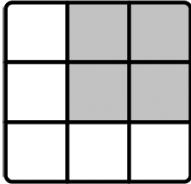
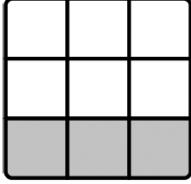
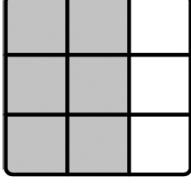
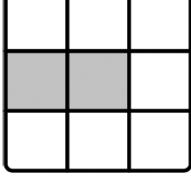
	Expresión	Forma
	arr[:2,1:]	(2,2)
	arr[2] arr[2, :] arr[2:, :]	(3,) (3,) (1,3)
	arr[:, :2]	(3,2)
	arr[1, :2] arr[1:2, :2]	(2,) (1,2)

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 pseudoaleatorios

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
<code>abs, fabs</code>	Calcula el valor absoluto elemento a elemento para valores enteros, de punto flotante o complejos.
<code>sqrt</code>	Calcula la raíz cuadrada de cada elemento (equivalente a <code>arr ** 0.5</code>).
<code>square</code>	Calcula el cuadrado de cada elemento (equivalente a <code>arr ** 2</code>).
<code>exp</code>	Calcula el exponente e^x de cada elemento.
<code>log, log10, log2, log1p</code>	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, sin, 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 ~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
minimum, fmin	Mínimo elemento a elemento; fmin ignora NaN.
mod	Módulo elemento a elemento (resto de la división).
copysign	Copia el signo de los valores del segundo argumento al primero.
greater, greater_equal, less, less_equal, equal, not_equal	Realiza comparaciones elemento a elemento, produciendo un array booleano (equivalente a las operaciones $>$, \geq , $<$, \leq , $==$, \neq).
logical_and	Calcula elemento a elemento el valor de verdad de la operación lógica AND ($\&$).
logical_or	Calcula elemento a elemento el valor de verdad de la operación lógica OR ($ $).
logical_xor	Calcula elemento a elemento el valor de verdad de la operación lógica XOR (\wedge).

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.

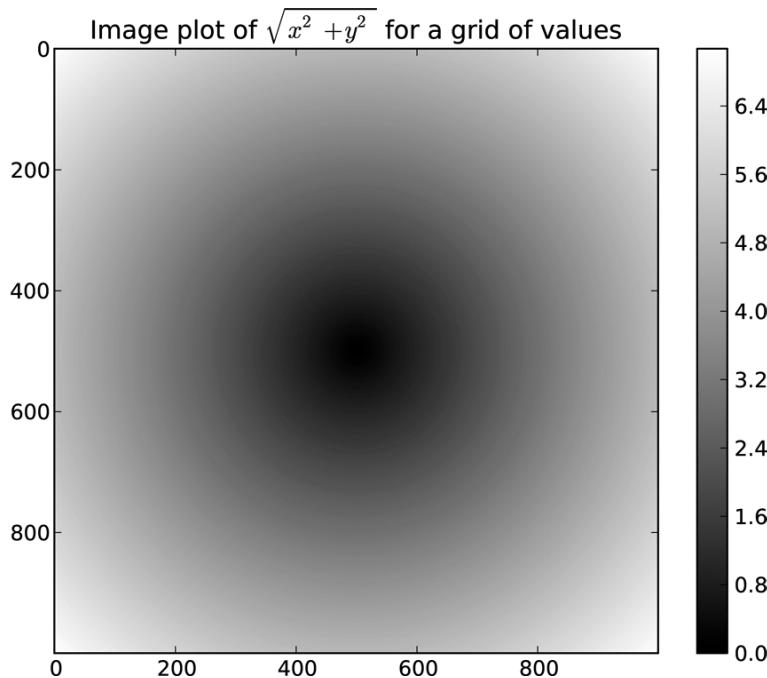


Figura 4.3. Representación de una función evaluada en una cuadrícula.

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 <code>Nan</code>) 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 un 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, 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])`

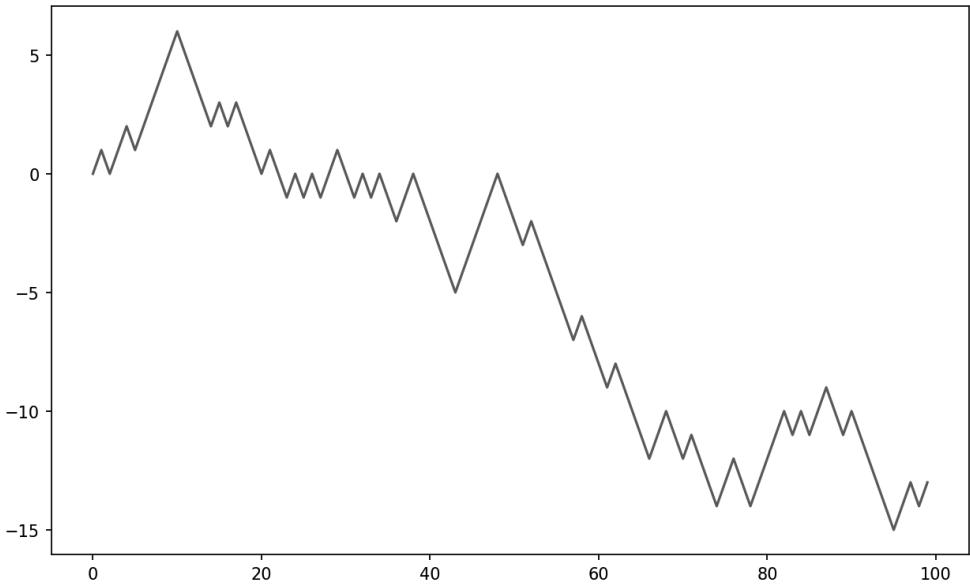


Figura 4.4. Un sencillo camino aleatorio.

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 todos 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
append()	Concatena con objetos índice adicionales, produciendo un nuevo índice.
difference()	Calcula la diferencia de conjuntos como un índice.
intersection()	Calcula la intersección de conjuntos.
union()	Calcula la unión de conjuntos.
isin()	Calcula un array booleano indicando si cada valor está contenido en la colección pasada.
delete()	Calcula un nuevo índice con el elemento del índice <i>i</i> borrado.
drop()	Calcula un nuevo índice borrando los valores pasados.
insert()	Calcula un nuevo índice insertando un elemento en el índice <i>i</i> .
is_monotonic	Devuelve True si cada elemento es mayor o igual que el elemento anterior.
is_unique	Devuelve True si el índice no tiene valores duplicados.
unique()	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                               NaN  
dtype: float64
```

Para series ordenadas, como, por ejemplo, las series temporales, quizá sea más interesante interpolar o llenar 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                               blue  
2                               purple  
4                               yellow  
dtype: object
```

```
In [104]: obj3.reindex(np.arange(6), method="ffill")  
Out[104]:
```

```
0                               blue  
1                               blue  
2                               purple  
3                               purple  
4                               yellow  
5                               yellow  
dtype: object
```

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 llenar.
<code>tolerance</code>	Cuando se rellena hacia delante o hacia atrás, el hueco de tamaño máximo (en distancia numérica absoluta) a llenar para coincidencias inexactas.
<code>level</code>	Coincide con el índice sencillo a nivel del índice jerárquico (<code>MultiIndex</code>); 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 1  
0 2  
1 3  
dtype: int64
```

In [140]: obj2.iloc[[0, 1, 2]]

Out[140]:

```
a 1  
b 2  
c 3  
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 1  
b 5  
c 5  
dtype: int64
```



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
three
Name: Colorado, dtype: int64
```

Después realizaremos algunas selecciones similares con enteros utilizando iloc:

```
In [157]: data.iloc[2]
Out[157]:
one
two
three
four
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
one
two
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 llenar 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]:
```

```
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 empataido.
"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           NaN  
d          -0.275  
dtype: float64
```

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          b  
two          d  
dtype: object
```

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", "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 metodo `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
read_csv	Carga datos delimitados de un archivo, una URL o un objeto de tipo archivo; usa la coma como delimitador predeterminado.
read_fwf	Lee datos en formato de columna de anchura fija (es decir, sin delimitadores).
read_clipboard	Variación de read_csv que lee datos del portapapeles; es útil para convertir tablas a partir de páginas web.
read_excel	Lee datos tabulares de un archivo XLS o XLSX de Excel.
read_hdf	Lee archivos HDF5 escritos por pandas.
read_html	Lee todas las tablas encontradas en el documento HTML dado.

Función	Descripción
read_json	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).
read_feather	Lee el formato de archivo binario Feather.
read_orc	Lee el formato de archivo binario ORC de Apache.
read_parquet	Lee el formato de archivo binario Parquet de Apache.
read_pickle	Lee un objeto almacenado por pandas empleando el formato pickle de Python.
read_sas	Lee un conjunto de datos SAS almacenado en uno de los formatos de almacenamiento personalizado del sistema SAS.
read_spss	Lee un archivo de datos creado por SPSS.
read_sql	Lee los resultados de una consulta SQL (utilizando SQLAlchemy).
read_sql_table	Lee una tabla SQL completa (utilizando SQLAlchemy); equivale a usar una consulta que lo selecciona todo en la tabla mediante read_sql.
read_stata	Lee un conjunto de datos de un formato de archivo Stata.
read_xml	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 argumentosopcionales. 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, quizás 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]:
```

message	a	b	c	d
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]:
```

key1	key2	value1	value2
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
path	Cadena de texto que indica una ubicación en el sistema de archivos, una URL o un objeto de tipo archivo.
sep o delimiter	Secuencia de caracteres o expresión regular que se emplea para dividir campos en cada fila.
header	Número de fila a utilizar como nombres de columna; por defecto es 0 (primera fila), pero debería ser None si no hay fila de encabezado.
index_col	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.
names	Lista de nombres de columna para el resultado.
skiprows	Número de filas al comienzo del archivo que hay que ignorar o lista de números de fila (empezando por 0) que hay que saltar.
na_values	Secuencia de valores para reemplazar por NA. Se añaden a la lista predeterminada a menos que se pase keep_default_na=False.
keep_default_na	Si se utiliza la lista de valores NA predeterminada o no (True por defecto).
comment	Carácter o caracteres para dividir comentarios al final de las líneas.
parse_dates	Intenta analizar datos en datetime; es False por defecto. Si es True, 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).
keep_date_col	Si se unen columnas para analizar la fecha, mantiene las columnas unidas; es False por defecto.
converters	Diccionario que contiene un número o nombre de columna asignado a funciones (es decir, {"foo": f} aplicaría la función f a todos los valores de la columna "foo").
dayfirst	Cuando se analizan fechas potencialmente ambiguas, se trata como si tuvieran formato internacional (por ejemplo, 7/6/2012 -> 7 de junio de 2012); es False por defecto.
date_parser	Función que se emplea para analizar fechas.
nrows	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   O
[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.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

muchas 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 dialecto 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 <code>True</code> , 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 <code>quoting</code> está fijado en <code>csv.QUOTE_NONE</code> ; 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		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]:
```

	0	a	b	c	d	message
0	0	1	2	3	4	hello
1	1	5	6	7	8	world
2	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 quizás 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": "MDU6TGFiZWwxMDY5MzUxMTM=", "url": "https..."}]
2 [{"id": 134699, "node_id": "MDU6TGFiZWwxMzQ20Tk=", "url": "https://api...."}]
3 [{"id": 233160, "node_id": "MDU6TGFiZWwyMzMxNjA=", "url": "https://api...."}]
4 [{"id": 32815646, "node_id": "MDU6TGFiZWwzMjgxNTY0Ng==", "url": "https:..."}]
.. .
25 [{"id": 76811, "node_id": "MDU6TGFiZWw3NjgxMQ==", "url": "https://api.g..."}]
26 [{"id": 76812, "node_id": "MDU6TGFiZWw3NjgxMg==", "url": "https://api.g..."}]
27 [{"id": 233160, "node_id": "MDU6TGFiZWwyMzMxNjA=", "url": "https://api...."}]
28 [{"id": 233160, "node_id": "MDU6TGFiZWwyMzMxNjA=", "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 0x7fdfd73b69c0>

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 0x7fdfd73a00c0>
```

```
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
cursor.description])
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>	Filtrá 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 llenar 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 llenar valores faltantes.
<code>method</code>	Método de interpolación: puede ser "bfill" (relleno hacia atrás) o "ffill" (relleno hacia delante); el valor predeterminado es <code>None</code> .
<code>axis</code>	Eje que llenar ("index" o "columns"); el valor predeterminado es <code>axis="index"</code> .
<code>limit</code>	Para lleno hacia delante o hacia atrás, máximo número de períodos consecutivos que llenar.

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 +  
[“two”],  
....: “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 duplicated 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

duplicated y 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 quizás 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]:
```

```
0                  pig
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                                         3.0
dtype: float64
```

Para utilizar un sustituto distinto para cada valor, pasamos una lista de sustitutos:

```
In [68]: data.replace([-999, -1000], [np.nan, 0])
Out[68]:
```

```
1.0
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]:
```

```
1.0
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
```

```
Out[80]:
```

```
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35, 60], (35, 60], (25, 35])  
Length: 12
```

```
Categories (4, interval[int64, right]): [(18, 25] < (25, 35] < (35, 60] < (60, 100)]
```

El objeto que pandas devuelve es un objeto Categorical especial. El resultado obtenido describe los contenedores calculados por `pandas.cut`. Cada contenedor está identificado mediante un tipo de valor de intervalo especial (único en pandas) que contiene los límites inferior y superior de cada contenedor:

```
In [81]: age_categories.codes  
Out[81]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1],  
dtype=int8)
```

```
In [82]: age_categories.categories  
Out[82]: IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]], dtype='interval  
[int64, right]')
```

```
In [83]: age_categories.categories[0]  
Out[83]: Interval(18, 25, closed='right')
```

```
In [84]: pd.value_counts(age_categories)  
Out[84]:
```

(18, 25]	5
(25, 35]	3
(35, 60]	3
(60, 100]	1
dtype:	int64

Podemos observar que `pd.value_counts(categories)` son los recuentos de contenedores para el resultado de `pandas.cut`.

En la representación de cadena de texto de un intervalo, un paréntesis significa que el lado está abierto (exclusivo), mientras que el corchete significa que está cerrado (inclusivo). Se puede cambiar el lado que está cerrado pasando `right=False`:

```
In [85]: pd.cut(ages, bins, right=False)
Out[85]:
[[18, 25), [18, 25), [25, 35), [25, 35), [18, 25), ..., [25,
35), [60, 100), [35, 60), [35, 60), [25, 35)]
Length: 12

Categories (4, interval[int64, left]): [[18, 25) < [25, 35)
< [35, 60) < [60, 100)]
```

Se puede anular el etiquetado de contenedor predeterminado basado en intervalos pasando una lista o un array a la opción `labels`:

```
In [86]: group_names = ["Youth", "YoungAdult", "MiddleAged",
"Senior"]

In [87]: pd.cut(ages, bins, labels=group_names)
Out[87]:
['Youth', 'Youth', 'Youth', 'YoungAdult', 'Youth', ...,
'YoungAdult', 'Senior', 'MiddleAged', 'MiddleAged', 'YoungAdult']
Length: 12

Categories (4, object): ['Youth' < 'YoungAdult' <
'MiddleAged' < 'Senior']
```

Si se pasa un número entero de contenedores a `pandas.cut` en lugar de bordes explícitos de contenedor, calculará contenedores de la misma longitud basándose en los valores mínimo y máximo de los datos. Veamos el caso de unos datos distribuidos uniformemente divididos en cuartos:

```
In [88]: data = np.random.uniform(size=20)

In [89]: pd.cut(data, 4, precision=2)
Out[89]:
[(0.34, 0.55], (0.34, 0.55], (0.76, 0.97], (0.76, 0.97],
(0.34, 0.55], ..., (0.34, 0.55], (0.34, 0.55], (0.55, 0.76],
(0.34, 0.55], (0.12, 0.34)]
```

```
Length: 20
Categories (4, interval[float64, right]): [(0.12, 0.34] <
(0.34, 0.55] < (0.55, 0.76] <
(0.76, 0.97]]
```

La opción `precision=2` limita la precisión decimal a dos dígitos.

Una función estrechamente relacionada, `pandas.qcut`, pone los datos en contenedores basándose en cuantiles de muestra. Dependiendo de la distribución de los datos, utilizar `pandas.cut` no dará normalmente como resultado que cada contenedor tenga el mismo número de puntos de datos. Como lo que hace `pandas.qcut` es usar cuantiles de muestra, se obtienen contenedores más o menos del mismo tamaño:

```
In [90]: data = np.random.standard_normal(1000)

In [91]: quartiles = pd.qcut(data, 4, precision=2)

In [92]: quartiles
Out[92]:
[(-0.026, 0.62], (0.62, 3.93], (-0.68, -0.026], (0.62,
3.93], (-0.026, 0.62], ...
, (-0.68, -0.026], (-0.68, -0.026], (-2.96, -0.68], (0.62,
3.93], (-0.68, -0.026])
Length: 1000
Categories (4, interval[float64, right]): [(-2.96, -0.68] <
(-0.68, -0.026] < (-0.026, 0.62] <
(0.62, 3.93]]

In [93]: pd.value_counts(quartiles)
Out[93]:
(-2.96, -0.68]                      250
(-0.68, -0.026]                      250
(-0.026, 0.62]                       250
(0.62, 3.93]                          250
dtype: int64
```

De forma similar a pandas.cut, se pueden pasar también cuantiles propios (números entre 0 y 1, inclusive):

```
In [94]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.]).value_counts()
Out[94]:
```

(-2.9499999999999997, -1.187]	100
(-1.187, -0.0265]	400
(-0.0265, 1.286]	400
(1.286, 3.928]	100
dtype:	int64

Volveremos a pandas.cut y pandas.qcut más adelante en este capítulo, en la parte dedicada a la agregación y a las operaciones de grupo, ya que estas funciones de discretización son especialmente útiles para análisis de cuantil y de grupo.

Detección y filtrado de valores atípicos

Filtrar o transformar outliers o valores atípicos es en gran parte cuestión de aplicar operaciones de arrays. Veamos un dataframe con datos normalmente distribuidos:

```
In [95]: data = pd.DataFrame(np.random.standard_normal((1000, 4)))
```

```
In [96]: data.describe()
Out[96]:
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.049091	0.026112	-0.002544	-0.051827
std	0.996947	1.007458	0.995232	0.998311
min	-3.645860	-3.184377	-3.745356	-3.428254
25%	-0.599807	-0.612162	-0.687373	-0.747478
50%	0.047101	-0.013609	-0.022158	-0.088274
75%	0.756646	0.695298	0.699046	0.623331

```
max           2.653656      3.525865      2.735527      3.366626
```

Supongamos que queremos encontrar valores en una de las columnas que sean mayores de 3 en valor absoluto:

```
In [97]: col = data[2]
```

```
In [98]: col[col.abs() > 3]
```

```
Out[98]:
```

```
41 -3.399312  
136 -3.745356
```

```
Name: 2, dtype: float64
```

Para seleccionar todas las filas con un valor superior a 3 o -3 , se puede emplear el método `any` en un dataframe booleano:

```
In [99]: data[(data.abs() > 3).any(axis="columns")]  
Out[99]:
```

	0	1	2	3
41	0.457246	-0.025907	-3.399312	-0.974657
60	1.951312	3.260383	0.963301	1.201206
136	0.508391	-0.196713	-3.745356	-1.520113
235	-0.242459	-3.056990	1.918403	-0.578828
258	0.682841	0.326045	0.425384	-3.428254
322	1.179227	-3.184377	1.369891	-1.074833
544	-3.548824	1.553205	-2.186301	1.277104
635	-0.578093	0.193299	1.397822	3.366626
782	-0.207434	3.525865	0.283070	0.544635
803	-3.645860	0.255475	-0.549574	-1.907459

Los paréntesis en torno a `data.abs() > 3` son necesarios para llamar al método `any` en el resultado de la operación de comparación.

Los valores se pueden configurar en base a estos criterios. Este código limita los valores que quedan fuera del intervalo -3 a 3 :

```
In [100]: data[data.abs() > 3] = np.sign(data) * 3
```

```
In [101]: data.describe()  
Out[101]:
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.050286	0.025567	-0.001399	-0.051765
std	0.992920	1.004214	0.991414	0.995761
min	-3.000000	-3.000000	-3.000000	-3.000000
25%	-0.599807	-0.612162	-0.687373	-0.747478
50%	0.047101	-0.013609	-0.022158	-0.088274
75%	0.756646	0.695298	0.699046	0.623331
max	2.653656	3.000000	2.735527	3.000000

La sentencia `np.sign(data)` produce valores 1 y -1 según si los valores de data son positivos o negativos:

```
In [102]: np.sign(data).head()  
Out[102]:
```

	0	1	2	3
0	-1.0	1.0	-1.0	1.0
1	1.0	-1.0	1.0	-1.0
2	1.0	1.0	1.0	-1.0
3	-1.0	-1.0	1.0	-1.0
4	-1.0	1.0	-1.0	-1.0

Permutación y muestreo aleatorio

La permutación (reordenación aleatoria) de una serie o de las filas de un dataframe es posible utilizando la función `numpy.random.permutation`. Llamar a `permutation` con la longitud del eje que se desea permutar produce un array de enteros que indica el nuevo orden:

```
In [103]: df = pd.DataFrame(np.arange(5 * 7).reshape((5, 7)))
```

```
In [104]: df  
Out[104]:
```

```
          0   1   2   3   4   5   6  
0       0   1   2   3   4   5   6  
1       7   8   9  10  11  12  13  
2      14  15  16  17  18  19  20  
3      21  22  23  24  25  26  27  
4      28  29  30  31  32  33  34
```

```
In [105]: sampler = np.random.permutation(5)
```

```
In [106]: sampler  
Out[106]: array([3, 1, 4, 2, 0])
```

Ese array se puede utilizar después en indexación basada en `iloc` o en la función equivalente `take`:

```
In [107]: df.take(sampler)  
Out[107]:
```

```
          0   1   2   3   4   5   6  
3       21  22  23  24  25  26  27  
1       7   8   9  10  11  12  13  
4       28  29  30  31  32  33  34  
2       14  15  16  17  18  19  20  
0       0   1   2   3   4   5   6
```

```
In [108]: df.iloc[sampler]  
Out[108]:
```

```
          0   1   2   3   4   5   6  
3       21  22  23  24  25  26  27  
1       7   8   9  10  11  12  13  
4       28  29  30  31  32  33  34  
2       14  15  16  17  18  19  20  
0       0   1   2   3   4   5   6
```

Invocando `take` con `axis="columns"` podemos también seleccionar una permutación de las columnas:

```
In [109]: column_sampler = np.random.permutation(7)
```

```
In [110]: column_sampler  
Out[110]: array([4, 6, 3, 2, 1, 0, 5])
```

```
In [111]: df.take(column_sampler, axis="columns")  
Out[111]:
```

	4	6	3	2	1	0	5
0	4	6	3	2	1	0	5
1	11	13	10	9	8	7	12
2	18	20	17	16	15	14	19
3	25	27	24	23	22	21	26
4	32	34	31	30	29	28	33

Para seleccionar un subconjunto aleatorio sin reemplazo (la misma fila no puede aparecer dos veces), podemos utilizar el método sample con objetos Series y DataFrame:

```
In [112]: df.sample(n=3)  
Out[112]:
```

	0	1	2	3	4	5	6
2	14	15	16	17	18	19	20
4	28	29	30	31	32	33	34
0	0	1	2	3	4	5	6

Para generar una muestra con reemplazo (y permitir así opciones de repetición), pasamos replace=True a sample:

```
In [113]: choices = pd.Series([5, 7, -1, 6, 4])  
  
In [114]: choices.sample(n=10, replace=True)  
Out[114]:
```

2	-1
0	5
3	6
1	7
4	4
0	5

```
4          4  
0          5  
4          4  
4          4  
dtype:    int64
```

Calcular variables dummy o indicadoras

Otro tipo de transformación para aplicaciones de modelado estadístico o aprendizaje automático es convertir una variable categórica en una matriz dummy o indicadora. Si una columna de un dataframe tiene k valores distintos, se podría derivar una matriz o un dataframe con k columnas conteniendo unos y ceros. En pandas tenemos una función `pandas.get_dummies` para hacer esto, aunque uno mismo puede también idear una. Veamos un dataframe de ejemplo:

```
In [115]: df = pd.DataFrame({"key": ["b", "b", "a", "c",  
"a", "b"],  
.....: "data1": range(6)})  
  
In [116]: df  
Out[116]:
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	b	5

```
In [117]: pd.get_dummies(df["key"])  
Out[117]:
```

	a	b	c
0	0	1	0
1	0	1	0
2	1	0	0
3	0	0	1

4		1	0	0
5		0	1	0

En algunos casos, quizá interese añadir un prefijo a las columnas en el dataframe indicador, que después se puede combinar con los demás datos. `pandas.get_dummies` tiene un argumento de prefijo para hacer esto:

```
In [118]: dummies = pd.get_dummies(df[“key”], prefix=“key”)
```

```
In [119]: df_with_dummy = df[["data1"]].join(dummies)
```

```
In [120]: df_with_dummy
```

```
Out[120]:
```

	data1	key_a	key_b	key_c
0	0	0	1	0
1	1	0	1	0
2	2	1	0	0
3	3	0	0	1
4	4	1	0	0
5	5	0	1	0

En el siguiente capítulo explicaré con más detalle el método `DataFrame.join`.

Si una fila de un dataframe pertenece a varias categorías, tenemos que utilizar otro enfoque para crear las variables indicadoras. Echemos un vistazo al conjunto de datos MovieLens 1M, que investigaremos a fondo en el capítulo 13:

```
In [121]: mnames = [“movie_id”, “title”, “genres”]
```

```
In [122]: movies = pd.read_table(“datasets/movielens/movies.dat”, sep=“::”,  
.....: header=None, names=mnames, engine=“python”)
```

```
In [123]: movies[:10]
```

```
Out[123]:
```

movie_id	title	genres
----------	-------	--------

0	1	Toy Story (1995)	Animation Children's Comedy
1	2	Jumanji (1995)	Adventure Children's Fantasy
2	3	Grumpier Old Men (1995)	Comedy Romance
3	4	Waiting to Exhale (1995)	Comedy Drama
4	5	Father of the Bride Part II (1995)	Comedy
5	6	Heat (1995)	Action Crime Thriller
6	7	Sabrina (1995)	Comedy Romance
7	8	Tom and Huck (1995)	Adventure Children's
8	9	Sudden Death (1995)	Action
9	10	GoldenEye (1995)	Action Adventure Thriller

pandas ha implementado un método especial `str.get_dummies` del objeto Series (los métodos que empiezan por `str.` se tratan con más detalle más adelante en este capítulo, en la sección «Manipulación de cadenas de texto»), que maneja esta situación de varias membresías de grupo codificadas como una cadena de texto delimitada:

```
In [124]: dummies = movies["genres"].str.get_dummies("|")
```

```
In [125]: dummies.iloc[:10, :6]
```

```
Out[125]:
```

	Action	Adventure	Animation	Children's	Comedy	Crime
0	0	0	1	1	1	0
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	0
4	0	0	0	0	1	0
5	1	0	0	0	0	1
6	0	0	0	0	1	0
7	0	1	0	1	0	0
8	1	0	0	0	0	0
9	1	1	0	0	0	0

Después, como antes, podemos combinar esto con `movies` añadiendo delante un “`Genre_`” a los nombres de columna del dataframe `dummies` con el método `add_prefix`:

```
In [126]: movies_windic = movies.join(dummies.add_prefix("Genre_"))
```

```
In [127]: movies_windic.iloc[0]  
Out[127]:
```

movie_id	1
title	Toy Story (1995)
genres	Animation Children's Comedy
Genre_Action	0
Genre_Adventure	0
Genre_Animation	1
Genre_Children's	1
Genre_Comedy	1
Genre_Crime	0
Genre_Documentary	0
Genre_Drama	0
Genre_Fantasy	0
Genre_Film-Noir	0
Genre_Horror	0
Genre_Musical	0
Genre_Mystery	0
Genre_Romance	0
Genre_Sci-Fi	0
Genre_Thriller	0
Genre_War	0
Genre_Western	0
Name: 0, dtype: object	



Para datos mucho más grandes, este método de construir variables indicadoras con múltiples membresías no es especialmente rápido. Sería mejor escribir una función de menor nivel que escribiera directamente en un array NumPy, y después envolviera el resultado en un dataframe.

Una receta útil para aplicaciones estadísticas es combinar `pandas.get_dummies` con una función de discretización como `pandas.cut`:

```
In [128]: np.random.seed(12345) # para que el ejemplo se pueda reutilizar
```

```
In [129]: values = np.random.uniform(size=10)
```

```
In [130]: values
```

```
Out[130]:
```

```
array([0.9296,  0.3164,  0.1839,  0.2046,  0.5677,  0.5955,
       0.9645,  0.6532,
       0.7489,  0.6536])
```

```
In [131]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]
```

```
In [132]: pd.get_dummies(pd.cut(values, bins))
```

```
Out[132]:
```

	(0.0, 0.2]	(0.2, 0.4]	(0.4, 0.6]	(0.6, 0.8]	(0.8, 1.0]	
0	0	0	0	0	0	1
1	0	1	0	0	0	0
2	1	0	0	0	0	0
3	0	1	0	0	0	0
4	0	0	1	0	0	0
5	0	0	1	0	0	0
6	0	0	0	0	0	1
7	0	0	0	1	0	0
8	0	0	0	1	0	0
9	0	0	0	0	1	0

Hablaremos de nuevo sobre `pandas.get_dummies` en la sección «Creación de variables indicadoras para modelado», más adelante en este capítulo.

7.3 Tipos de datos de extensión



Este es un tema nuevo y avanzado que muchos usuarios de pandas no tienen por qué conocer, pero lo presento aquí para completar la información, porque usaré los tipos de datos de extensión y haré referencia a ellos en distintos momentos de los siguientes capítulos.

pandas se construyó inicialmente en base a las capacidades presentes en NumPy, una librería de cálculo de arrays empleada fundamentalmente para trabajar con datos numéricos. Muchos conceptos de pandas, como los datos ausentes, se implementaron empleando lo que estaba disponible en NumPy, pero intentando al mismo tiempo maximizar la compatibilidad entre librerías que usaban tanto NumPy como pandas.

Construir sobre NumPy dio lugar a una serie de deficiencias, como por ejemplo:

- La manipulación de datos ausentes para ciertos tipos de datos numéricos, como los enteros y los booleanos, estaba incompleta. Como resultado de ello, cuando se introducían datos ausentes dentro de dichos datos, pandas convertía el tipo de datos en `float64` y utilizaba `np.nan` para representar los valores nulos. Esto agravó la situación, introduciendo problemas sutiles en muchos algoritmos de pandas.
- Los conjuntos de datos que tenían muchas cadenas de texto resultaban caros desde el punto de vista computacional y utilizaban mucha memoria.
- Algunos tipos de datos, como los intervalos de tiempo, los `timedelta` y los `timestamp` con zonas horarias, no se podían soportar eficazmente sin utilizar arrays de objetos Python, que resultan caros desde el punto de vista computacional.

Recientemente, pandas ha desarrollado un sistema de tipos de extensión, que permite añadir nuevos tipos de datos incluso aunque NumPy no los soporte de forma nativa. Estos nuevos tipos de datos se pueden tratar como datos de primera clase junto con los procedentes de los arrays NumPy.

Veamos un ejemplo en el que creamos una serie de enteros con un valor ausente:

```
In [133]: s = pd.Series([1, 2, 3, None])
```

```
In [134]: s  
Out[134]:
```

```
0          1.0
1          2.0
2          3.0
3          NaN
dtype: float64
```

```
In [135]: s.dtype
Out[135]: dtype('float64')
```

Principalmente por razones de compatibilidad con versiones anteriores, los objetos Series usan el comportamiento heredado de utilizar un tipo de datos float64 y np.nan para el valor ausente. Pero podríamos crear esta otra serie empleando pandas.Int64Dtype:

```
In [136]: s = pd.Series([1, 2, 3, None],
dtype=pd.Int64Dtype())
```

```
In [137]: s
Out[137]:
```

```
0          1
1          2
2          3
3          <NA>
dtype: Int64
```

```
In [138]: s.isna()
Out[138]:
```

```
0      False
1      False
2      False
3      True
dtype: bool
```

```
In [139]: s.dtype
Out[139]: Int64Dtype()
```

La salida <NA> indica que falta un valor para un array de tipo de extensión; dicho valor emplea el valor centinela especial pandas.NA:

```
In [140]: s[3]
Out[140]: <NA>
```

```
In [141]: s[3] is pd.NA
Out[141]: True
```

También podríamos haber utilizado la abreviatura “Int64” en lugar de pd.Int64Dtype() para especificar el tipo. Las mayúsculas son necesarias, ya que, de otro modo, será un tipo de no extensión basado en NumPy:

```
In [142]: s = pd.Series([1, 2, 3, None], dtype="Int64")
```

pandas tiene también un tipo de extensión especializado para datos de cadena de texto que no utiliza arrays de objeto NumPy (requiere la librería pyarrow, que quizá sea necesario instalar por separado):

```
In [143]: s = pd.Series(['one', 'two', None, 'three'],
dtype=pd.StringDtype())
```

```
In [144]: s
Out[144]:
```

0	one
1	two
2	<NA>
3	three
dtype:	string

Estos arrays de cadena de texto usan generalmente mucha menos memoria y suelen ser computacionalmente más eficientes para realizar operaciones con grandes conjuntos de datos.

Otro tipo de extensión importante es categorical, del que hablaremos con más detalle en la sección «Datos categóricos», en este mismo capítulo. En la tabla 7.3 se puede consultar una lista razonablemente completa de tipos de extensión disponibles en el momento en que se escribe este libro.

Los tipos de extensión se pueden pasar al método astype del objeto Series, facilitando la conversión como parte del proceso de limpieza de datos:

```
In [145]: df = pd.DataFrame({"A": [1, 2, None, 4],
.....:     "B": ["one", "two", "three", None],
.....:     "C": [False, None, False, True]})
```

```
In [146]: df
Out[146]:
```

	A	B	C
0	1.0	one	False
1	2.0	two	None
2	NaN	three	False
3	4.0	None	True

```
In [147]: df["A"] = df["A"].astype("Int64")
```

```
In [148]: df["B"] = df["B"].astype("string")
```

```
In [149]: df["C"] = df["C"].astype("boolean")
```

```
In [150]: df
Out[150]:
```

	A	B	C
0	1	one	False
1	2	two	<NA>
2	<NA>	three	False
3	4	<NA>	True

Tabla 7.3. Tipos de datos de extensión de pandas.

Tipo de extensión	Descripción
BooleanDtype	Datos booleanos anulables, usa "boolean" cuando se pasa como cadena de texto.

CategoricalDtype	Tipo de datos categórico, usa "category" cuando se pasa como cadena de texto.
DatetimeTZDtype	Datetime con zona horaria.
Float32Dtype	Punto flotante anulable de 32 bits, usa "Float32" cuando se pasa como cadena de texto.
Float64Dtype	Punto flotante anulable de 64 bits, usa "Float64" cuando se pasa como cadena de texto.
Int8Dtype	Entero con signo anulable de 8 bits, usa "Int8" cuando se pasa como cadena de texto.
Int16Dtype	Entero con signo anulable de 16 bits, usa "Int16" cuando se pasa como cadena de texto.
Int32Dtype	Entero con signo anulable de 32 bits, usa "Int32" cuando se pasa como cadena de texto.
Int64Dtype	Entero con signo anulable de 64 bits, usa "Int64" cuando se pasa como cadena de texto.
Uint8Dtype	Entero sin signo anulable de 8 bits, usa "UInt8" cuando se pasa como cadena de texto.
Uint16Dtype	Entero sin signo anulable de 16 bits, usa "UInt16" cuando se pasa como cadena de texto.
Uint32Dtype	Entero sin signo anulable de 32 bits, usa "UInt32" cuando se pasa como cadena de texto.
Uint64Dtype	Entero sin signo anulable de 64 bits, usa "UInt64" cuando se pasa como cadena de texto.

7.4 Manipulación de cadenas de texto

Python ha sido durante mucho tiempo un lenguaje de manipulación de datos sin procesar muy popular, debido en parte a lo fácil que es utilizarlo para procesar cadenas de texto y texto en general. La mayoría de las operaciones con texto resultan muy sencillas gracias a los métodos internos del objeto cadena de texto. Pero para manipulaciones de texto y coincidencia de patrones más complejas, pueden ser necesarias las expresiones regulares. También pandas ofrece la posibilidad de aplicar de

manera concisa expresiones regulares y de cadena de texto a arrays de datos completos, gestionando además los molestos valores ausentes.

Métodos de objeto de cadena de texto internos de Python

En muchas aplicaciones de scripting y procesado de cadenas de texto, los métodos internos de cadenas de texto son suficientes. A modo de ejemplo, una cadena de texto separada por comas se puede dividir en partes con `split`:

```
In [151]: val = "a,b, guido"
```

```
In [152]: val.split(",")
Out[152]: ['a', 'b', ' guido']
```

`split` suele combinarse con `strip` para quitar los espacios en blanco (incluyendo los saltos de línea):

```
In [153]: pieces = [x.strip() for x in val.split(",")]
```

```
In [154]: pieces
Out[154]: ['a', 'b', 'guido']
```

Estas subcadenas de texto se podrían concatenar con el signo de dos puntos como delimitador utilizando `addition`:

```
In [155]: first, second, third = pieces
```

```
In [156]: first + "::" + second + ":" + third
Out[156]: 'a::b::guido'
```

Pero este no es un método genérico que resulte práctico. Una forma más rápida y más de estilo Python es pasar una lista o tupla al método `join` en la cadena de texto “::”:

```
In [157]: ":".join(pieces)
Out[157]: 'a::b::guido'
```

Otros métodos tienen que ver con localizar subcadenas de texto. Utilizar la palabra clave `in` de Python es la mejor manera de detectar una subcadena de texto, aunque también pueden utilizarse `index` y `find`:

```
In [158]: "guido" in val  
Out[158]: True
```

```
In [159]: val.index(",")  
Out[159]: 1
```

```
In [160]: val.find(":")  
Out[160]: -1
```

Hay que tener en cuenta que la diferencia entre `find` e `index` es que `index` produce una excepción si no se encuentra la cadena de texto (frente a devolver `-1`):

```
In [161]: val.index(":")
```

```
ValueError      Traceback (most recent call last)
```

```
<ipython-input-161-bea4c4c30248> in <module>  
--> 1 val.index(":")
```

```
ValueError: substring not found
```

También `count` devuelve el número de apariciones de una subcadena de texto en particular:

```
In [162]: val.count(",")  
Out[162]: 2
```

`replace` sustituirá las apariciones de un patrón por otro. También se suele usar para eliminar patrones pasando una cadena de texto vacía:

```
In [163]: val.replace(", ", "::")  
Out[163]: 'a::b:: guido'
```

```
In [164]: val.replace(", ", "")
```

```
Out[164]: 'ab guido'
```

La tabla 7.4 muestra un listado de algunos de los métodos de cadena de texto de Python.

También se pueden emplear las expresiones regulares con muchas de estas operaciones, como veremos.

Tabla 7.4. Métodos de cadena de texto internos de Python.

Método	Descripción
count	Devuelve el número de apariciones no superpuestas de una subcadena de texto en la cadena de texto.
endswith	Devuelve True si la cadena de texto termina con un sufijo.
startswith	Devuelve True si la cadena de texto empieza con un prefijo.
join	Usa una cadena de texto como delimitador para concatenar una secuencia de otras cadenas de texto.
index	Devuelve el índice inicial de la primera aparición de la subcadena de texto pasada si se encuentra en la cadena de texto; si no se encuentra, da un ValueError.
find	Devuelve la posición del primer carácter de la primera aparición de la subcadena de texto en la cadena de texto; es igual index, pero devuelve -1 si no se encuentra.
rfind	Devuelve la posición del primer carácter de la última aparición de la subcadena de texto en la cadena de texto; devuelve -1 si no se encuentra.
replace	Sustituye las apariciones de la cadena de texto por otra cadena de texto.
strip, rstrip, lstrip	Quita espacios en blanco, incluyendo nuevas líneas en ambos lados, en el lado derecho o en el lado izquierdo, respectivamente.
split	Divide la cadena de texto en listas de subcadenas utilizando el delimitador pasado.
lower	Convierte caracteres alfabéticos a minúsculas.
upper	Convierte caracteres alfabéticos a mayúsculas.
casefold	Convierte caracteres alfabéticos a minúsculas, y convierte cualquier combinación de caracteres variables específicos de una determinada región a una forma común comparable.
ljust,	Justifica a la izquierda o a la derecha, respectivamente; llena el lado contrario de la

`rjust`

cadena de texto con espacios (o con otro carácter de relleno) para devolver una cadena de texto con una anchura mínima.

Expresiones regulares

Las expresiones regulares ofrecen una manera flexible de buscar en un texto patrones de cadenas de texto o encontrar coincidencias (algo con frecuencia más complejo). Una expresión sencilla, denominada habitualmente regex, es una cadena de texto formada de acuerdo con el lenguaje de las expresiones regulares. El módulo `re` interno de Python es el responsable de aplicar las expresiones regulares a las cadenas de texto; aquí daré varios ejemplos de su uso.



El arte de escribir expresiones regulares podría dar lugar a un capítulo completo por sí solo, de ahí que se salga del alcance de este libro. Existen muchos tutoriales y guías de referencia excelentes en Internet y en otros libros.

Las funciones del módulo `re` entran en tres categorías: coincidencia de patrones, reemplazo y división. Por supuesto, todas están relacionadas; un regex describe un patrón para localizar en el texto, que después se puede utilizar para muchas cosas. Veamos un ejemplo sencillo: supongamos que queremos dividir una cadena de texto con un número variable de caracteres de espacio en blanco (tabuladores, espacios y nuevas líneas).

El regex que describe uno o más caracteres de espacio en blanco es `\s+`:

```
In [165]: import re  
In [166]: text = "foo bar\tbaz \tqux"  
In [167]: re.split(r"\s+", text)  
Out[167]: ['foo', 'bar', 'baz', 'qux']
```

Cuando llamamos a `re.split(r"\s+", text)`, la expresión regular primero se compila y después se llama a su método `split` en el texto pasado. Uno mismo puede compilar el regex con `re.compile`, formando un objeto regex reutilizable:

```
In [168]: regex = re.compile(r"\s+")
```

```
In [169]: regex.split(text)
Out[169]: ['foo', 'bar', 'baz', 'qux']
```

Pero si lo que queríamos era obtener una lista de todos los patrones que coinciden con el regex, podemos usar el método `findall`:

```
In [170]: regex.findall(text)
Out[170]: ['', '\t ', '\t']
```



Para evitar escapes no deseados con \ en una expresión regular, usamos literales de cadena de texto sin procesar como `r"C:\x"`, en lugar del equivalente “`C:\\x`”.

Crear un objeto `regex` con `re.compile` es muy recomendable si la intención es aplicar la misma expresión a muchas cadenas de texto; hacerlo ahorrará ciclos de la CPU.

`match` y `search` están estrechamente relacionados con `findall`. Mientras `findall` devuelve todas las coincidencias de una cadena de texto, `search` devuelve únicamente la primera coincidencia. Con una mayor rigidez, `match` solo encuentra coincidencias al principio de la cadena de texto. Como ejemplo menos trivial, veamos un bloque de texto y una expresión regular capaz de identificar la mayoría de las direcciones de correo electrónico:

```
text = """Dave dave@google.com
Steve steve@gmail.com

Rob rob@gmail.com
Ryan ryan@yahoo.com"""
pattern = r"[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}"
# re.IGNORECASE hace que regex no reconozca las mayúsculas

regex = re.compile(pattern, flags=re.IGNORECASE)
```

Utilizando `findall` en el texto se obtiene una lista de las direcciones de correo electrónico:

```
In [172]: regex.findall(text)
Out[172]:
```

```
[ 'dave@google.com',
  'steve@gmail.com',
  'rob@gmail.com',
  'ryan@yahoo.com' ]
```

search devuelve un objeto de coincidencia especial para la primera dirección de correo electrónico del texto. Para el regex anterior, el objeto de coincidencia solo puede decírnos la posición inicial y final del patrón en la cadena de texto:

```
In [173]: m = regex.search(text)
```

```
In [174]: m
Out[174]: <re.Match object; span=(5, 20),
match='dave@google.com'>
```

```
In [175]: text[m.start():m.end()]
Out[175]: 'dave@google.com'
```

regex.match devuelve None, ya que coincidirá solo si el patrón aparece al principio de la cadena de texto:

```
In [176]: print(regex.match(text))
```

```
None
```

En relación con esto, sub devuelve una nueva cadena de texto con las apariciones del patrón reemplazadas por una nueva cadena de texto:

```
In [177]: print(regex.sub("REDACTED", text))
Dave REDACTED
```

```
Steve REDACTED
Rob REDACTED
```

```
Ryan REDACTED
```

Supongamos que queremos encontrar direcciones de correo electrónico y dividir simultáneamente cada dirección en sus tres componentes: nombre de

usuario, nombre de dominio y sufijo de dominio. Para ello, ponemos paréntesis alrededor de las partes del patrón a segmentar:

```
In [178]: pattern = r"([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})"
```

```
In [179]: regex = re.compile(pattern, flags=re.IGNORECASE)
```

Un objeto de coincidencia producido por este regex modificado devuelve una tupla de los componentes del patrón con su método `groups`:

```
In [180]: m = regex.match("wesm@bright.net")
```

```
In [181]: m.groups()
```

```
Out[181]: ('wesm', 'bright', 'net')
```

`.findall` devuelve una lista de tuplas cuando el patrón tiene grupos:

```
In [182]: regex.findall(text)
```

```
Out[182]:
```

```
[('dave', 'google', 'com'),
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]
```

`sub` tiene acceso también a grupos en cada coincidencia utilizando símbolos especiales como `\1` y `\2`. El símbolo `\1` corresponde al primer grupo localizado, `\2` corresponde con el segundo, y así sucesivamente:

```
In [183]: print(regex.sub(r"Username: \1, Domain: \2, Suffix: \3", text))
Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

Python ofrece mucha más información relacionada con las expresiones regulares, la mayor parte de la cual queda fuera del alcance de este libro. La tabla 7.5 ofrece un breve resumen.

Tabla 7.5. Métodos de expresiones regulares.

Método	Descripción
<code>findall</code>	Devuelve todos los patrones de coincidencia no superpuestos de una cadena de texto como una lista.
<code>finditer</code>	Igual que <code>findall</code> , pero devuelve un iterador.
<code>match</code>	Localiza un patrón al comienzo de la cadena de texto y divide opcionalmente sus componentes en grupos; si el patrón coincide, devuelve un objeto de coincidencia, si no, devuelve <code>None</code> .
<code>search</code>	Busca una cadena de texto que coincida con un patrón, devolviendo un objeto de coincidencia si la encuentra; a diferencia de <code>match</code> , la coincidencia puede estar en cualquier punto de la cadena de texto en lugar de solo al principio.
<code>split</code>	Divide una cadena de texto en partes en cada aparición del patrón.
<code>sub,</code> <code>subn</code>	Reemplaza todas (<code>sub</code>) o las primeras <code>n</code> (<code>subn</code>) apariciones del patrón en la cadena de texto con la expresión de reemplazo; usa los símbolos <code>\1</code> , <code>\2</code> , etc. para referirse a los elementos del grupo de coincidencia en la cadena de texto de reemplazo.

Funciones de cadena de texto en pandas

Limpiar un conjunto de datos desordenado para su análisis suele requerir mucha manipulación de cadenas de texto. Para complicar las cosas, una columna que contiene cadenas de texto suele tener datos ausentes:

```
In [184]: data = {"Dave": "dave@google.com", "Steve": "steve@gmail.com",
.....: "Rob": "rob@gmail.com", "Wes": np.nan}
```

```
In [185]: data = pd.Series(data)
```

```
In [186]: data
Out[186]:
```

Dave	dave@google.com
Steve	steve@gmail.com
Rob	rob@gmail.com
Wes	NaN

```
dtype: object

In [187]: data.isna()
Out[187]:
```

Dave	False
Steve	False
Rob	False
Wes	True
dtype:	bool

Pueden aplicarse métodos de cadena de texto y expresión regular (pasando una lambda u otra función) a cada valor empleando `data.map`, pero dará error en los valores NA (nulos). Para solucionar esto, el objeto Series tiene métodos orientados a arrays para operaciones con cadenas de texto que omiten y propagan los valores NA. Se puede acceder a ellos con el atributo `str` del objeto Series; por ejemplo, podríamos comprobar si cada dirección de correo electrónico contiene “gmail” con `str.contains`:

```
In [188]: data.str.contains("gmail")
Out[188]:
```

Dave	False
Steve	True
Rob	True
Wes	NaN
dtype:	object

Conviene observar que el `dtype` del resultado de esta operación es `object`. pandas tiene tipos de extensión que ofrecen un tratamiento especializado a las cadenas de texto y los datos enteros y booleanos que, hasta hace poco, tenían algunos problemas a la hora de trabajar con datos ausentes:

```
In [189]: data_as_string_ext = data.astype('string')
```

```
In [190]: data_as_string_ext  
Out[190]:
```

```
Dave          dave@google.com  
Steve         steve@gmail.com  
Rob           rob@gmail.com  
Wes           <NA>  
dtype:        string
```

```
In [191]: data_as_string_ext.str.contains("gmail")  
Out[191]:
```

```
Dave          False  
Steve         True  
Rob           True  
Wes           <NA>  
dtype:        boolean
```

Los tipos de extensión se tratan con más detalle en la sección «Tipos de datos de extensión».

Las expresiones regulares también pueden utilizarse junto con cualquier opción de `re`, como `IGNORECASE`:

```
In [192]: pattern = r"([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})"
```

```
In [193]: data.str.findall(pattern, flags=re.IGNORECASE)  
Out[193]:
```

```
Dave          [(dave, google, com)]  
Steve         [(steve, gmail, com)]  
Rob           [(rob, gmail, com)]  
Wes           NaN  
dtype:        object
```

Hay un par de maneras de recuperar elementos vectorizados. Se puede utilizar o bien `str.get` o indexar en el atributo `str`:

```
In [194]: matches = data.str.findall(pattern,  
flags=re.IGNORECASE).str[0]
```

```
In [195]: matches  
Out[195]:
```

```
Dave (dave, google, com)  
Steve (steve, gmail, com)  
Rob (rob, gmail, com)  
Wes NaN  
dtype: object
```

```
In [196]: matches.str.get(1)  
Out[196]:
```

```
Dave google  
Steve gmail  
Rob gmail  
Wes NaN  
dtype: object
```

También se pueden cortar las cadenas de texto de forma similar utilizando esta sintaxis:

```
In [197]: data.str[:5]  
Out[197]:
```

```
Dave dave@  
Steve steve  
Rob rob@g  
Wes NaN  
dtype: object
```

El método str.extract devolverá los grupos capturados de una expresión regular como un dataframe:

```
In [198]: data.str.extract(pattern, flags=re.IGNORECASE)
```

Out[198] :

	0	1	2
Dave	dave	google	com
Steve	steve	gmail	com
Rob	rob	gmail	com
Wes	NaN	NaN	NaN

Véanse en la siguiente tabla más métodos de cadena de texto de pandas.

Tabla 7.6. Listado parcial de métodos de cadena de texto del objeto Series.

Método	Descripción
cat	Concatena cadenas de texto elemento a elemento con un delimitador opcional.
contains	Devuelve un array booleano si cada cadena contiene un patrón o regex.
count	Cuenta las apariciones de un patrón.
extract	Usa una expresión regular con grupos para extraer una o varias cadenas de texto de una serie de cadenas de texto; el resultado será un dataframe con una columna por grupo.
endswith	Equivalente a <code>x.endswith(pattern)</code> para cada elemento.
startswith	Equivalente a <code>x.startswith(pattern)</code> para cada elemento.
findall	Calcula la lista de todas las apariciones del patrón o regex para cada cadena de texto.
get	Obtiene el índice de dentro de cada elemento (recupera el elemento i-ésimo).
isalnum	Equivalente a la función <code>str.isalnum</code> interna.
isalpha	Equivalente a la función <code>str.isalpha</code> interna.
isdecimal	Equivalente a la función <code>str.isdecimal</code> interna.
isdigit	Equivalente a la función <code>str.isdigit</code> interna.
islower	Equivalente a la función <code>str.islower</code> interna.
isnumeric	Equivalente a la función <code>str.isnumeric</code> interna.
isupper	Equivalente a la función <code>str.isupper</code> interna.

Método	Descripción
join	Une cadenas de texto de cada elemento de la serie con un separador pasado.
len	Calcula la longitud de cada cadena de texto.
lower, upper	Convierte las mayúsculas y minúsculas; equivalente a <code>x.lower()</code> o <code>x.upper()</code> para cada elemento.
match	Usa <code>re.match</code> con la expresión regular pasada en cada elemento, devolviendo <code>True</code> o <code>False</code> si coincide.
pad	Añade espacios en blanco a la izquierda, derecha o a ambos lados de las cadenas de texto.
center	Equivalente a <code>pad(side="both")</code> .
repeat	Duplica valores (por ejemplo, <code>s.str.repeat(3)</code> es equivalente a <code>x * 3</code> para cada cadena de texto).
replace	Reemplaza las apariciones del patrón o regex con alguna otra cadena de texto.
slice	Corta cada cadena de texto de la serie.
split	Divide cadenas de texto según un delimitador o una expresión regular.
strip	Quita espacios en blanco de ambos lados, incluyendo nuevas líneas.
rstrip	Quita espacios en blanco del lado derecho.
lstrip	Quita espacios en blanco del lado izquierdo.

7.5 Datos categóricos

Esta sección presenta el tipo `Categorical` de pandas. Mostraré cómo se puede lograr un mejor rendimiento y uso de la memoria en algunas operaciones en pandas empleando este tipo. También voy a introducir algunas herramientas que pueden ayudar a utilizar datos categóricos en aplicaciones de estadística y aprendizaje automático.

Antecedentes y motivación

Una columna de una tabla puede contener con frecuencia instancias repetidas de un conjunto más pequeño de valores diferentes. Ya hemos visto funciones como `unique` y `value_counts`, que nos permiten extraer los distintos valores de un array y calcular sus frecuencias, respectivamente:

```
In [199]: values = pd.Series(['apple', 'orange', 'apple',
.....: 'apple'] * 2)
```

```
In [200]: values
Out[200]:
```

```
          apple
0           apple
1           orange
2           apple
3           apple
4           apple
5           orange
6           apple
7           apple
dtype: object
```

```
In [201]: pd.unique(values)
Out[201]: array(['apple', 'orange'], dtype=object)
```

```
In [202]: pd.value_counts(values)
Out[202]:
```

```
apple 6
orange 2
dtype: int64
```

Muchos sistemas de datos (para almacenar datos, realizar cálculos estadísticos u otros usos) han desarrollado métodos especializados para representar los datos con valores repetidos para un almacenamiento y cálculo más eficaces. En almacenamiento de datos, una buena práctica es utilizar las denominadas tablas de dimensiones, que contienen los distintos valores y almacenan las observaciones principales como claves enteras haciendo referencia a la tabla de dimensiones:

```
In [203]: values = pd.Series([0, 1, 0, 0] * 2)
```

```
In [204]: dim = pd.Series(['apple', 'orange'])
```

```
In [205]: values
```

```
Out[205]:
```

0	0
1	1
2	0
3	0
4	0
5	1
6	0
7	0
dtype:	int64

```
In [206]: dim
```

```
Out[206]:
```

0	apple
1	orange
dtype:	object

Podemos usar el método take para restaurar la serie original de cadenas de texto:

```
In [207]: dim.take(values)
```

```
Out[207]:
```

0	apple
1	orange
0	apple
0	apple
0	apple
1	orange
0	apple
0	apple
dtype:	object

Esta representación como enteros se llama representación categórica o codificada como un diccionario. Al conjunto de los distintos valores se le puede denominar las categorías, el diccionario o los niveles de los datos. En este libro utilizaremos los términos categórico y categorías. Los valores enteros que hacen referencia a las categorías se llaman códigos de categoría o simplemente códigos.

La representación categórica puede dar lugar a notables mejoras en el rendimiento cuando se realizan análisis. También se pueden llevar a cabo transformaciones con las categorías, dejando los códigos sin modificar. Algunas transformaciones de ejemplo que se pueden hacer a un coste relativamente bajo son las siguientes:

- Renombrar categorías.
 - Añadir una nueva categoría sin cambiar el orden o la posición de las categorías existentes.

Tipo de extensión Categorical en pandas

pandas tiene un tipo de extensión categorical especial para guardar datos que usa la representación categórica basada en enteros o codificación. Se trata de una conocida técnica de compresión de datos para datos con muchas apariciones de valores similares, y puede ofrecer un rendimiento bastante más rápido con un menor uso de la memoria, especialmente con datos de cadenas de texto.

Veamos la serie de ejemplo ya empleada anteriormente:

```
In [208]: fruits = ['apple', 'orange', 'apple', 'apple'] * 2
```

```
In [209]: N = len(fruits)
```

```
In [210]: rng = np.random.default_rng(seed=12345)
```

```
In [211]: df = pd.DataFrame({'fruit': fruits,
```

```
.....: 'basket_id': np.arange(N),
```

```
....:         'count': rng.integers(3, 15, size=N),
....:         'weight': rng.uniform(0, 4, size=N)},
....: columns=['basket_id', 'fruit', 'count', 'weight'])
```

In [212]: df

Out[212]:

	basket_id	fruit	count	weight
0	0	apple	11	1.564438
1	1	orange	5	1.331256
2	2	apple	12	2.393235
3	3	apple	6	0.746937
4	4	apple	5	2.691024
5	5	orange	12	3.767211
6	6	apple	10	0.992983
7	7	apple	11	3.795525

Aquí, df['fruit'] es un array de objetos de cadena de texto de Python.

Podemos convertirlo en categórico llamando a:

In [213]: fruit_cat = df['fruit'].astype('category')

In [214]: fruit_cat

Out[214]:

0	apple
1	orange
2	apple
3	apple
4	apple
5	orange
6	apple
7	apple

Name: fruit, dtype: category

Categories (2, object): ['apple', 'orange']

Los valores para fruit_cat son ahora una instancia de pandas.Categorical, al que se puede acceder mediante el atributo .array:

```
In [215]: c = fruit_cat.array  
In [216]: type(c)  
Out[216]: pandas.core.arrays.categorical.Categorical
```

El objeto Categorical tiene atributos categories y codes:

```
In [217]: c.categories  
Out[217]: Index(['apple', 'orange'], dtype='object')  
  
In [218]: c.codes  
Out[218]: array([0, 1, 0, 0, 0, 1, 0, 0], dtype=int8)
```

Es posible acceder a ellos fácilmente mediante el atributo de acceso cat, que se explicará pronto en la sección «Métodos categóricos».

Un truco útil para obtener un mapeado entre códigos y categorías es:

```
In [219]: dict(enumerate(c.categories))  
Out[219]: {0: 'apple', 1: 'orange'}
```

Se puede convertir una columna de un dataframe en categórica asignando el resultado convertido:

```
In [220]: df['fruit'] = df['fruit'].astype('category')  
In [221]: df["fruit"]  
Out[221]:
```

0	apple
1	orange
2	apple
3	apple
4	apple
5	orange
6	apple
7	apple

Name: fruit, dtype: category

```
Categories (2, object): ['apple', 'orange']
```

También se puede crear pandas.Categorical directamente a partir de otros tipos de secuencias de Python:

```
In [222]: my_categories = pd.Categorical(['foo', 'bar',  
'baz', 'foo', 'bar'])
```

```
In [223]: my_categories  
Out[223]:  
['foo', 'bar', 'baz', 'foo', 'bar']
```

```
Categories (3, object): ['bar', 'baz', 'foo']
```

Si hubiéramos obtenido datos codificados categóricos de otra fuente, podríamos utilizar el constructor alternativo `from_codes`:

```
In [224]: categories = ['foo', 'bar', 'baz']
```

```
In [225]: codes = [0, 1, 2, 0, 0, 1]
```

```
In [226]: my_cats_2 = pd.Categorical.from_codes(codes,  
categories)
```

```
In [227]: my_cats_2  
Out[227]:
```

```
['foo', 'bar', 'baz', 'foo', 'foo', 'bar']
```

```
Categories (3, object): ['foo', 'bar', 'baz']
```

A menos que se especifique explícitamente, las conversiones categóricas no asumen una ordenación determinada de las categorías. Por lo tanto, el array `categories` puede estar en un orden distinto, dependiendo de cómo estén ordenados los datos de entrada. Cuando se utiliza `from_codes` o cualquiera de los otros constructores, se puede indicar que las categorías tienen un orden significativo:

```
In [228]: ordered_cat = pd.Categorical.from_codes(codes,  
categories,  
.....: ordered=True)
```

```
In [229]: ordered_cat
```

```
Out[229]:
```

```
['foo', 'bar', 'baz', 'foo', 'foo', 'bar']
```

```
Categories (3, object): ['foo' < 'bar' < 'baz']
```

La salida [foo < bar < baz] indica que 'foo' precede a 'bar' en la ordenación, y así sucesivamente. Una instancia categórica no ordenada se puede ordenar con `as_ordered`:

```
In [230]: my_cats_2.as_ordered()
```

```
Out[230]:
```

```
['foo', 'bar', 'baz', 'foo', 'foo', 'bar']
```

```
Categories (3, object): ['foo' < 'bar' < 'baz']
```

Como última anotación, los datos categóricos no tienen que ser cadenas de texto, aunque solo hayamos visto aquí ejemplos con este tipo de datos. Un array categórico puede estar formado por cualquier tipo de valor inmutable.

Cálculos con variables categóricas

Utilizar `Categorical` en pandas comparado con la versión no codificada (como un array de cadenas de texto) se comporta generalmente del mismo modo. Algunas partes de pandas, como la función `groupby`, funcionan mejor cuando trabajan con variables categóricas. Hay también varias funciones que pueden emplear la bandera `ordered`. Veamos unos cuantos datos numéricos aleatorios y empleemos la función de discretización `pandas.qcut`. Esto devuelve `pandas.Categorical`; usamos `pandas.cut` anteriormente en el libro, pero apenas vimos los detalles de cómo funcionan las variables categóricas:

```
In [231]: rng = np.random.default_rng(seed=12345)
```

```
In [232]: draws = rng.standard_normal(1000)
```

```
In [233]: draws[:5]
```

```
Out[233]: array([-1.4238,  1.2637, -0.8707, -0.2592,
 -0.0753])
```

Calculemos una discretización de cuartil de estos datos y extraigamos algunas estadísticas:

```
In [234]: bins = pd.qcut(draws, 4)

In [235]: bins
Out[235]:
[(-3.121, -0.675], (0.687, 3.211], (-3.121, -0.675],
(-0.675, 0.0134], (-0.675, 0.0134], ..., (0.0134, 0.687],
(0.0134, 0.687], (-0.675, 0.0134], (0.0134, 0.687], (-0.675,
0.0134])
Length: 1000
Categories (4, interval[float64, right]): [(-3.121, -0.675]
< (-0.675, 0.0134] <
(0.0134, 0.687] <
(0.687, 3.211]]
```

Aunque sirven, los cuartiles de muestra exactos pueden ser menos útiles para producir un informe que los nombres de cuartil. Podemos conseguir esto con el argumento `labels` para `qcut`:

```
In [236]: bins = pd.qcut(draws, 4, labels=['Q1', 'Q2', 'Q3',
 'Q4'])

In [237]: bins
Out[237]:
['Q1', 'Q4', 'Q1', 'Q2', 'Q2', ..., 'Q3', 'Q3', 'Q2', 'Q3',
 'Q2']
Length: 1000
Categories (4, object): ['Q1' < 'Q2' < 'Q3' < 'Q4']

In [238]: bins.codes[:10]
Out[238]: array([0, 3, 0, 1, 1, 0, 0, 2, 2, 0], dtype=int8)
```

La variable categórica etiquetada como `bins` no contiene información sobre los bordes de contenedor de los datos, de modo que podemos usar `groupby` para extraer ciertas estadísticas de resumen:

```
In [239]: bins = pd.Series(bins, name='quartile')
```

```
In [240]: results = (pd.Series(draws)
```

```
.....:     .groupby(bins)
.....:     .agg(['count', 'min', 'max'])
.....:     .reset_index()
```

```
In [241]: results
```

```
Out[241]:
```

	quartile	count	min	max
0	Q1	250	-3.119609	-0.678494
1	Q2	250	-0.673305	0.008009
2	Q3	250	0.018753	0.686183
3	Q4	250	0.688282	3.211418

La columna 'quartile' del resultado conserva la información categórica original de bins, incluyendo la ordenación:

```
In [242]: results['quartile']
```

```
Out[242]:
```

0	Q1
1	Q2
2	Q3
3	Q4

```
Name: quartile, dtype: category
```

```
Categories (4, object): ['Q1' < 'Q2' < 'Q3' < 'Q4']
```

Mejor rendimiento con variables categóricas

Al principio de la sección dije que los tipos categóricos pueden mejorar el rendimiento y el uso de la memoria, de modo que veamos algunos ejemplos. Supongamos algunas series con 10 millones de elementos y un pequeño número de categorías diferentes:

```
In [243]: N = 10_000_000
```

```
In [244]: labels = pd.Series(['foo', 'bar', 'baz', 'qux'] * (N // 4))
```

Ahora convertimos labels a categórico:

```
In [245]: categories = labels.astype('category')
```

Podemos observar que ahora labels usa bastante más memoria que categories:

```
In [246]: labels.memory_usage(deep=True)
Out[246]: 600000128
```

```
In [247]: categories.memory_usage(deep=True)
Out[247]: 10000540
```

La conversión a categoría no es gratuita, por supuesto, pero tiene un coste único:

```
In [248]: %time _ = labels.astype('category')
CPU times: user 469 ms, sys: 106 ms, total: 574 ms
Wall time: 577 ms
```

Las operaciones GroupBy pueden ser notablemente más rápidas con variables categóricas porque los algoritmos subyacentes usan los códigos basados en enteros, en lugar de un array de cadenas de texto. Aquí comparamos el rendimiento de value_counts(), que emplea internamente la maquinaria de GroupBy:

```
In [249]: %timeit labels.value_counts()
840 ms +- 10.9 ms per loop (mean +- std. dev. of 7 runs, 1
loop each)
```

```
In [250]: %timeit categories.value_counts()
30.1 ms +- 549 us per loop (mean +- std. dev. of 7 runs, 10
loops each)
```

Métodos categóricos

Las series que contienen datos categóricos tienen varios métodos especiales similares a los métodos de cadena de texto especializados Series.str. Esto proporciona asimismo un cómodo acceso a las categorías y los códigos. Veamos la siguiente serie:

```
In [251]: s = pd.Series(['a', 'b', 'c', 'd'] * 2)
```

```
In [252]: cat_s = s.astype('category')
```

```
In [253]: cat_s
```

```
Out[253]:
```

0	a
1	b
2	c
3	d
4	a
5	b
6	c
7	d
dtype:	category

Categories (4, object): ['a', 'b', 'c', 'd']

El atributo de acceso especial cat permite el acceso a los métodos categóricos:

```
In [254]: cat_s.cat.codes
```

```
Out[254]:
```

0	0
1	1
2	2
3	3
4	0
5	1
6	2
7	3
dtype:	int8

```
In [255]: cat_s.cat.categories  
Out[255]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

Supongamos que sabemos que el conjunto real de categorías para estos datos va más allá de los cuatro valores observados en los datos. Podemos utilizar el método `set_categories` para cambiarlos:

```
In [256]: actual_categories = ['a', 'b', 'c', 'd', 'e']  
In [257]: cat_s2 = cat_s.cat.set_categories(actual_categories)  
In [258]: cat_s2  
Out[258]:
```

```
0          a  
1          b  
2          c  
3          d  
4          a  
5          b  
6          c  
7          d  
dtype: category
```

```
Categories (5, object): ['a', 'b', 'c', 'd', 'e']
```

Aunque parece que los datos no se han modificado, las nuevas categorías se verán reflejadas en las operaciones que las usen. Por ejemplo, `value_counts` respeta las categorías, si están presentes:

```
In [259]: cat_s.value_counts()  
Out[259]:
```



```
a    2  
b    2  
c    2  
d    2  
dtype: int64
```

```
In [260]: cat_s2.value_counts()  
Out[260]:
```

a	2
b	2
c	2
d	2
e	0
dtype:	int64

En grandes conjuntos de datos, las variables categóricas se utilizan habitualmente como una herramienta conveniente para ahorrar memoria y obtener un mejor rendimiento. Una vez filtrado un dataframe o una serie grande, muchas de las categorías pueden no aparecer en los datos. Para solucionar esto, podemos emplear el método `remove_unused_categories` para quitar las categorías no observadas:

```
In [261]: cat_s3 = cat_s[cat_s.isin(['a', 'b'])]
```

```
In [262]: cat_s3  
Out[262]:
```

0	a
1	b
4	a
5	b
dtype:	category

```
Categories (4, object): ['a', 'b', 'c', 'd']
```

```
In [263]: cat_s3.cat.remove_unused_categories()  
Out[263]:
```

0	a
1	b
4	a
5	b
dtype:	category

```
Categories (2, object): ['a', 'b']
```

Véase en la tabla 7.7 un listado de los métodos categóricos disponibles.

Tabla 7.7. Métodos categóricos para objetos Series en pandas.

Método	Descripción
add_categories	Añade nuevas categorías (no utilizadas) al final de las ya existentes.
as_ordered	Hace que las categorías estén ordenadas.
as_unordered	Hace que las categorías estén desordenadas
remove_categories	Elimina categorías, estableciendo los valores eliminados en nulos.
removed_unused_categories	Elimina cualquier valor de categoría que no aparezca en los datos.
rename_categories	Reemplaza categorías con el conjunto indicado de nuevos nombres de categoría; no puede cambiar el número de categorías.
reorder_categories	Se comporta como rename_categories, pero puede cambiar también el resultado de haber ordenado las categorías.
set_categories	Reemplaza las categorías con el conjunto indicado de categorías nuevas; puede añadir o eliminar categorías.

Creación de variables indicadoras para modelado

Cuando se utilizan herramientas para estadística o aprendizaje automático, a menudo se transforman los datos categóricos en variables indicadoras, proceso también conocido como codificación one-hot. Implica la creación de un objeto DataFrame con una columna para cada categoría distinta; estas columnas contienen unos para las apariciones de una determinada categoría y ceros en otro caso.

Veamos el ejemplo anterior:

```
In [264]: cat_s = pd.Series(['a', 'b', 'c', 'd'] * 2,
                           dtype='category')
```

Como mencioné anteriormente en este capítulo, la función `pandas.get_dummies` convierte estos datos categóricos unidimensionales en un dataframe que contiene la variable indicadora:

```
In [265]: pd.get_dummies(cat_s)  
Out[265]:
```

	a	b	c	d
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	1	0	0	0
5	0	1	0	0
6	0	0	1	0
7	0	0	0	1

7.6 Conclusión

La preparación eficaz de los datos puede mejorar notablemente la productividad, pues permiten emplear más tiempo en el análisis de los datos y menos en prepararlos para su análisis. Hemos explorado distintas herramientas en este capítulo, pero la cobertura dada aquí no es ni mucho menos exhaustiva. En el siguiente capítulo abordaremos la funcionalidad de unión y agrupamiento de pandas.

Disputa de datos: unión, combinación y remodelación

En muchas aplicaciones, los datos pueden estar repartidos entre varios archivos o bases de datos u organizados de un modo que no resulta cómodo para analizarlos. Este capítulo se centra en herramientas que facilitan la combinación, unión y reordenación de los datos.

En primer lugar voy a presentar el concepto de indexación jerárquica en pandas, que se emplea mucho en algunas de estas operaciones. Después me adentraré en particular en las manipulaciones de datos.

El capítulo 13 ofrece varios ejemplos de aplicación de estas herramientas.

8.1 Indexación jerárquica

La indexación jerárquica es una característica importante de pandas que permite tener varios niveles de índice en un eje o, dicho de otro modo, proporciona una forma de trabajar con datos de muchas dimensiones en una forma de menor dimensión. Empecemos con un ejemplo sencillo: crear una serie con una lista de listas (o arrays) como índice:

```
In [11]: data = pd.Series(np.random.uniform(size=9),  
....:     index=[["a", "a", "a", "b", "b", "c", "c", "d",  
....:             "d"],  
....:             [1, 2, 3, 1, 3, 1, 2, 2, 3]])  
  
In [12]: data  
Out[12]:
```

```
a      1      0.929616
       2      0.316376
       3      0.183919
b      1      0.204560
       3      0.567725
c      1      0.595545
       2      0.964515
d      2      0.653177
       3      0.748907
```

dtype: float64

Aquí estamos viendo «en bonito» una serie con un objeto MultiIndex como índice. Los «huecos» en la visualización del índice significan «usar la etiqueta directamente encima».

```
In [13]: data.index
Out[13]:
```

```
MultiIndex([
              ('a', 1),
              ('a', 2),
              ('a', 3),
              ('b', 1),
              ('b', 3),
              ('c', 1),
              ('c', 2),
              ('d', 2),
              ('d', 3)],
             )
```

Con un objeto jerárquicamente indexado es posible aplicar la denominada indexación parcial, que permite seleccionar de manera concisa subconjuntos de los datos:

```
In [14]: data["b"]
Out[14]:
```

```
1          0.204560
3          0.567725
dtype: float64
```

```
In [15]: data["b":"c"]
Out[15]:
```

```
b      1          0.204560
      3          0.567725
c      1          0.595545
      2          0.964515
dtype: float64
```

```
In [16]: data.loc[["b", "d"]]
Out[16]:
```

```
b      1          0.204560
      3          0.567725
d      2          0.653177
      3          0.748907
dtype: float64
```

La selección es incluso posible en un nivel «interior». Aquí selecciono todos los valores que coinciden con el 2 en el segundo nivel de índice:

```
In [17]: data.loc[:, 2]
Out[17]:
```

```
a          0.316376
c          0.964515
d          0.653177
dtype: float64
```

La indexación jerárquica desempeña un papel de gran importancia en la remodelación de datos y en operaciones basadas en grupos, como, por

ejemplo, en la formación de una tabla dinámica. Por ejemplo, se pueden reordenar estos datos en un dataframe utilizando su método `unstack`:

```
In [18]: data.unstack()  
Out[18]:
```

	1	2	3
a	0.929616	0.316376	0.183919
b	0.204560	NaN	0.567725
c	0.595545	0.964515	NaN
d	NaN	0.653177	0.748907

La operación inversa a `unstack` es `stack`:

```
In [19]: data.unstack().stack()  
Out[19]:
```

a	1	0.929616
	2	0.316376
	3	0.183919
b	1	0.204560
	3	0.567725
c	1	0.595545
	2	0.964515
d	2	0.653177
	3	0.748907
dtype:	float64	

Exploraremos `stack` y `unstack` con más detalle en la sección posterior «Remodelación y transposición».

Con un dataframe, cualquier eje puede tener un índice jerárquico:

```
In [20]: frame = pd.DataFrame(np.arange(12).reshape((4, 3)),  
....:     index=[["a", "a", "b", "b"], [1, 2, 1, 2]],  
....:     columns=[["Ohio", "Ohio", "Colorado"],  
....:               ["Green", "Red", "Green"]])
```

```
In [21]: frame  
Out[21]:
```

	Ohio		Colorado
	Green	Red	Green
a 1	0	1	2
2	3	4	5
b 1	6	7	8
2	9	10	11

Los niveles jerárquicos pueden tener nombre (como una cadena de texto o cualquier objeto Python). Si lo tienen, aparece en el resultado visualizado:

```
In [22]: frame.index.names = ["key1", "key2"]
```

```
In [23]: frame.columns.names = ["state", "color"]
```

```
In [24]: frame
```

```
Out[24]:
```

state	color	key1	key2	Ohio	Red	Colorado
				Green		Green
a		1		0	1	2
		2		3	4	5
b		1		6	7	8
		2		9	10	11

Estos nombres sustituyen al atributo name, que únicamente se emplea con índices de un solo nivel.



Hay que tener en cuenta que los nombres de índice “state” y “color” no forman parte de las etiquetas de fila (los valores `frame.index`).

Se puede saber cuantos niveles tiene un índice accediendo a su atributo `nlevels`:

```
In [25]: frame.index.nlevels  
Out[25]: 2
```

Con indexación de columnas parcial se pueden seleccionar grupos de columnas de forma parecida:

```
In [26]: frame["Ohio"]  
Out[26]:
```

color	key1	key2	Green	Red
a	1	1	0	1
		2	3	4
b	1	1	6	7
		2	9	10

Un objeto MultiIndex puede crearse por sí mismo y después reutilizarse; las columnas del dataframe anterior con nombres de nivel también podrían haberse creado así:

```
pd.MultiIndex.from_arrays([["Ohio", "Ohio", "Colorado"],  
                           ["Green", "Red", "Green"]],  
                           names=["state", "color"])
```

Reordenación y clasificación de niveles

En ocasiones puede ser necesario modificar el orden de los niveles de un eje o clasificar los datos según los valores de un determinado nivel. El método swaplevel toma dos números o nombres de nivel y devuelve un objeto nuevo con los niveles intercambiados (pero, por lo demás, los datos quedan inalterados):

```
In [27]: frame.swaplevel("key1", "key2")  
Out[27]:
```

```
Ohio          Colorado
```

state	color		Green	Red	Green
key2	key1				
1	a	0	1		2
2	a	3	4		5
1	b	6	7		8
2	b	9	10		11

De forma predeterminada, `sort_index` clasifica los datos lexicográficamente, utilizando todos los niveles de índice, pero se puede tener la opción de emplear solamente un nivel o un subconjunto de niveles para realizar la clasificación pasando el argumento `level`. Por ejemplo:

```
In [28]: frame.sort_index(level=1)
Out[28]:
```

state	color		Ohio		Colorado
key1	key2		Green	Red	Green
a	1	0	1		2
b	1	3	4		5
a	2	6	7		8
b	2	9	10		11

```
In [29]: frame.swaplevel(0, 1).sort_index(level=0)
Out[29]:
```

state	color		Ohio		Colorado
key2	key1		Green	Red	Green
1	a	0	1		2
	b	3	4		5
2	a	6	7		8
	b	9	10		11



El rendimiento de la selección de datos es mucho mejor con objetos jerárquicamente indexados si el índice está clasificado lexicográficamente empezando por el nivel más externo, es decir, el resultado de llamar a `sort_index(level=0)` o a `sort_index()`.

Estadísticas de resumen por nivel

Muchas estadísticas descriptivas y de resumen en objetos DataFrame y Series tienen una opción `level`, en la que es posible especificar el nivel según el cual se desea agregar en un determinado eje. Recuperemos el dataframe anterior; podemos agregar por nivel en las filas o en las columnas, del siguiente modo:

```
In [30]: frame.groupby(level="key2").sum()  
Out[30]:
```

	Ohio		Colorado
state	Green	Red	Green
color			
key2			
1	6	8	10
2	12	14	16

```
In [31]: frame.groupby(level="color", axis="columns").sum()  
Out[31]:
```

		Green	Red
color			
key1	key2		
a	1	2	1
	2	8	4
b	1	14	7
	2	20	10

Hablaremos con más detalle de groupby en el capítulo 10.

Indexación con las columnas de un dataframe

No es raro querer utilizar una o varias columnas de un dataframe como índice de fila; la alternativa es que nos interese mover el índice de fila a las columnas del dataframe. Aquí tenemos un dataframe de ejemplo:

```
In [32]: frame = pd.DataFrame({"a": range(7), "b": range(7, 0, -1),
....: "c": ["one", "one", "one", "two", "two",
....: "two", "two"], "d": [0, 1, 2, 0, 1, 2, 3]})
```

```
In [33]: frame
Out[33]:
```

	a	b	c	d
0	0	7	one	0
1	1	6	one	1
2	2	5	one	2
3	3	4	two	0
4	4	3	two	1
5	5	2	two	2
6	6	1	two	3

La función `set_index` del objeto DataFrame creará un nuevo dataframe utilizando una o varias de sus columnas como índice:

```
In [34]: frame2 = frame.set_index(["c", "d"])
```

```
In [35]: frame2
Out[35]:
```

		a	b
c	d		
one	0	0	7
	1	1	6
	2	2	5
two	0	3	4
	1	4	3
	2	5	2

3

6

1

Las columnas se eliminan por defecto del dataframe, aunque se pueden dejar pasando drop=False a set_index:

```
In [36]: frame.set_index(["c", "d"], drop=False)
Out[36]:
```

		a	b	c	d
c	d				
one	0	0	7	one	0
	1	1	6	one	1
	2	2	5	one	2
two	0	3	4	two	0
	1	4	3	two	1
	2	5	2	two	2
	3	6	1	two	3

Por otro lado, reset_index hace lo contrario de set_index; los niveles del índice jerárquico se desplazan a las columnas:

```
In [37]: frame2.reset_index()
Out[37]:
```

	c	d	a	b
0	one	0	0	7
1	one	1	1	6
2	one	2	2	5
3	two	0	3	4
4	two	1	4	3
5	two	2	5	2
6	two	3	6	1

8.2 Combinación y fusión de conjuntos de datos

Los datos contenidos en objetos pandas se pueden combinar de distintas formas:

- `pandas.merge`: Conecta filas de dataframes según una o varias claves. Esto les resultará familiar a los usuarios de SQL u otras bases de datos relacionales, ya que implementa operaciones de unión de bases de datos.
- `pandas.concat`: Concatena o «apila» objetos a lo largo de un eje.
- `combine_first`: Une datos que se solapan para llenar valores ausentes en un objeto con valores de otro.

Haré referencia a cada una de estas funciones y daré varios ejemplos, pues se utilizarán en otros ejemplos del resto del libro.

Uniones de dataframes al estilo de una base de datos

Las operaciones de fusión o unión combinan conjuntos de datos enlazando filas mediante una o varias claves. Estas operaciones son especialmente importantes en bases de datos relacionales (basadas en SQL). La función `pandas.merge` de pandas es el principal punto de entrada para usar estos algoritmos con nuestros datos.

Empecemos con un ejemplo sencillo:

```
In [38]: df1 = pd.DataFrame({"key": ["b", "b", "a", "c",
"a", "a", "b"],
```

```
.....: "      data1": pd.Series(range(7), dtype="Int64")})
```

```
In [39]: df2 = pd.DataFrame({"key": ["a", "b", "d"],
```

```
.....: "data2": pd.Series(range(3), dtype="Int64")})
```

```
In [40]: df1
```

```
Out[40]:
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3

4	a	4
5	a	5
6	b	6

In [41]: df2

Out[41]:

	key	data2
0	a	0
1	b	1
2	d	2

Aquí estoy utilizando el tipo de extensión `Int64` de pandas para enteros anulables, que ya vimos en la sección «Tipos de datos de extensión» del capítulo 7.

Se trata de un ejemplo de una unión de muchos a uno; los datos de `df1` tienen varias filas con etiquetas a y b, mientras que `df2` solo tiene una fila para cada valor de la columna `key`. Llamando a `pandas.merge` con estos objetos, obtenemos lo siguiente:

In [42]: `pd.merge(df1, df2)`

Out[42]:

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0

No he especificado en qué columna hacer la unión. Si no se incluye esa información, `pandas.merge` usa los nombres de columna que se solapan como claves. No obstante, es una buena práctica especificarlo de manera explícita:

```
In [43]: pd.merge(df1, df2, on="key")
Out[43]:
```

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0

En general no se especifica el orden del resultado de las columnas en operaciones pandas.`merge`.

Si los nombres de columnas son distintos en cada objeto, se pueden especificar por separado:

```
In [44]: df3 = pd.DataFrame({"lkey": ["b", "b", "a", "c",
                                         "a", "a", "b"],
                               ....: "data1": pd.Series(range(7), dtype="Int64")})
```

```
In [45]: df4 = pd.DataFrame({"rkey": ["a", "b", "d"],
                               ....: "data2": pd.Series(range(3), dtype="Int64")})
```

```
In [46]: pd.merge(df3, df4, left_on="lkey", right_on="rkey")
Out[46]:
```

	lkey	data1	rkey	data2
0	b	0	b	1
1	b	1	b	1
2	b	6	b	1
3	a	2	a	0
4	a	4	a	0
5	a	5	a	0

Quizá algún lector se haya percatado de que los valores “c” y “d” y sus datos asociados no aparecen en el resultado. De forma predeterminada, pandas.merge realiza una unión “inner”; las claves del resultado son la intersección, o el conjunto común hallado en ambas tablas. Otras posibles opciones son “left”, “right” y “outer”. La unión externa toma la unión de las claves, combinando el efecto de aplicar las uniones de la izquierda y la derecha:

```
In [47]: pd.merge(df1, df2, how="outer")
Out[47]:
```

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0
6	c	3	<NA>
7	d	<NA>	2

```
In [48]: pd.merge(df3, df4, left_on="lkey", right_on="rkey",
how="outer")
Out[48]:
```

	lkey	data1	rkey	data2
0	b	0	b	1
1	b	1	b	1
2	b	6	b	1
3	a	2	a	0
4	a	4	a	0
5	a	5	a	0
6	c	3	NaN	<NA>
7	NaN	<NA>	d	2

En una unión externa, las filas de los objetos DataFrame de la izquierda o derecha que no coinciden con claves del otro dataframe aparecerán con valores NA en las columnas del otro dataframe para las filas no coincidentes.

La tabla 8.1 muestra un resumen de las opciones de how.

Tabla 8.1. Distintos tipos de unión con el argumento how.

Opción	Comportamiento
how="inner"	Usa solamente las combinaciones de claves observadas en ambas tablas.
how="left"	Usa todas las combinaciones de claves halladas en la tabla izquierda.
how="right"	Usa todas las combinaciones de claves halladas en la tabla derecha.
how="outer"	Usa todas las combinaciones de claves observadas en ambas tablas juntas.

Las fusiones de muchos a muchos forman el producto cartesiano de las claves coincidentes. Este es un ejemplo de ello:

```
In [49]: df1 = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "b"],
```

```
....:     "data1": pd.Series(range(6), dtype="Int64")})
```

```
In [50]: df2 = pd.DataFrame({"key": ["a", "b", "a", "b", "d"],
```

```
....:     "data2": pd.Series(range(5), dtype="Int64")})
```

```
In [51]: df1
```

```
Out[51]:
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3

4	a	4
5	b	5

In [52]: df2
Out[52]:

	key	data2
0	a	0
1	b	1
2	a	2
3	b	3
4	d	4

In [53]: pd.merge(df1, df2, on="key", how="left")
Out[53]:

	key	data1	data2
0	b	0	1
1	b	0	3
2	b	1	1
3	b	1	3
4	a	2	0
5	a	2	2
6	c	3	<NA>
7	a	4	0
8	a	4	2
9	b	5	1
10	b	5	3

Como había tres filas “b” en el dataframe de la izquierda y dos en el de la derecha, en el resultado hay seis filas “b”. El método de unión pasado al argumento de la palabra clave how solo afecta a los distintos valores de clave que aparecen en el resultado:

In [54]: pd.merge(df1, df2, how="inner")
Out[54]:

	key	data1	data2
--	-----	-------	-------

0	b	0	1
1	b	0	3
2	b	1	1
3	b	1	3
4	b	5	1
5	b	5	3
6	a	2	0
7	a	2	2
8	a	4	0
9	a	4	2

Para fusionar con varias claves, pasamos una lista de nombres de columna:

```
In [55]: left = pd.DataFrame({"key1": ["foo", "foo", "bar"],
....:     "key2": ["one", "two", "one"],
....:     "lval": pd.Series([1, 2, 3], dtype='Int64')})
```

```
In [56]: right = pd.DataFrame({"key1": ["foo", "foo", "bar",
"bar"],
....:     "key2": ["one", "one", "one", "two"],
....:     "rval": pd.Series([4, 5, 6, 7], dtype='Int64')})
```

```
In [57]: pd.merge(left, right, on=["key1", "key2"],
how="outer")
Out[57]:
```

	key1	key2	lval	rval
0	foo	one	1	4
1	foo	one	1	5
2	foo	two	2	<NA>
3	bar	one	3	6
4	bar	two	<NA>	7

Para determinar qué combinaciones de claves aparecerán en el resultado dependiendo de la elección del método de fusión, lo mejor es pensar en las

distintas claves como si formaran un array de tuplas que se utilizará como una sola clave de unión.



Cuando se unen columnas a columnas, los índices de los objetos DataFrame pasados se descartan. Si necesitamos conservar los valores de los índices, podemos usar `reset_index` para añadir el índice a las columnas.

Una última cuestión a tener en cuenta en las operaciones de fusión es el tratamiento de los nombres de columna que se superponen. Por ejemplo:

```
In [58]: pd.merge(left, right, on="key1")
Out[58]:
```

	key1	key2_x	lval	key2_y	rval
0	foo	one	1	one	4
1	foo	one	1	one	5
2	foo	two	2	one	4
3	foo	two	2	one	5
4	bar	one	3	one	6
5	bar	one	3	two	7

Aunque se puede resolver la superposición manualmente (la sección «Renombrar índices de eje» del capítulo 7 ofrece información al respecto), `pandas.merge` tiene una opción `suffixes` para especificar las cadenas de texto que se van a añadir a nombres que se superponen en los objetos DataFrame de la izquierda y derecha:

```
In [59]: pd.merge(left, right, on="key1", suffixes=("_left",
                                             "_right"))
Out[59]:
```

	key1	key2_left	lval	key2_right	rval
0	foo	one	1	one	4
1	foo	one	1	one	5
2	foo	two	2	one	4
3	foo	two	2	one	5
4	bar	one	3	one	6
5	bar	one	3	two	7

En la tabla 8.2 se pueden consultar los argumentos de pandas.merge. La siguiente sección trata la unión empleando el índice de fila del objeto DataFrame.

Tabla 8.2. Argumentos de la función pandas.merge.

Argumento	Descripción
<code>left</code>	Dataframe para fusionar por el lado izquierdo.
<code>right</code>	Dataframe para fusionar por el lado derecho.
<code>how</code>	Tipo de unión que se va a aplicar: puede ser "inner", "outer", "left" o "right"; el valor predeterminado es "inner".
<code>on</code>	Nombres de las columnas que se van a unir. Se debe encontrar en ambos objetos DataFrame. Si no se especifica o no se dan otras claves de unión, utilizará la intersección de los nombres de columnas de <code>left</code> y <code>right</code> como claves de unión.
<code>left_on</code>	Columnas del dataframe <code>left</code> para usar como claves de unión. Puede ser un solo nombre de columna o una lista de nombres de columnas.
<code>right_on</code>	Análogo a <code>left_on</code> para el dataframe <code>right</code> .
<code>left_index</code>	Utiliza el índice de fila de <code>left</code> como clave de unión (o claves, si es un objeto MultiIndex).
<code>right_index</code>	Análogo a <code>left_index</code> .
<code>sort</code>	Clasifica datos fusionados lexicográficamente según las claves de unión; es <code>False</code> por defecto.
<code>suffixes</code>	Tupla de valores de cadena de texto a añadir a nombres de columnas en caso de superposición; el valor predeterminado es (" <code>_x</code> ", " <code>_y</code> ") (por ejemplo, si existe "data" en ambos objetos DataFrame, aparecería en el resultado como "data_ <code>x</code> " y "data_ <code>y</code> ").
<code>copy</code>	Si es <code>False</code> , evita copiar datos en la estructura de datos resultante en algunos casos excepcionales; por defecto siempre copia.
<code>validate</code>	Verifica si la fusión es del tipo especificado, ya sea de uno a uno, de uno a muchos, o de muchos a muchos. Consulte en el docstring todos los detalles de las opciones.

Argumento	Descripción
indicator	Añade una columna especial <code>_merge</code> que indica de dónde procede cada fila; los valores serán "left_only", "right_only" o "both" según el origen de los datos combinados en cada fila.

Fusión según el índice

En algunos casos, la clave o claves de fusión de un dataframe se encontrarán en su índice (etiquetas de fila). En este caso, se puede pasar `left_index=True` o `right_index=True` (o ambos) para indicar que el índice debería usarse como clave para la unión:

```
In [60]: left1 = pd.DataFrame({"key": ["a", "b", "a", "a", "b", "c"],
```

```
.....:     "value": pd.Series(range(6), dtype="Int64")})
```

```
In [61]: right1 = pd.DataFrame({"group_val": [3.5, 7]}, index=["a", "b"])
```

```
In [62]: left1
Out[62]:
```

	key	value
0	a	0
1	b	1
2	a	2
3	a	3
4	b	4
5	c	5

```
In [63]: right1
Out[63]:
```

```
group_val
a 3.5
b 7.0
```

```
In [64]: pd.merge(left1, right1, left_on="key",
right_index=True)
Out[64]:
```

	key	value	group_val
0	a	0	3.5
2	a	2	3.5
3	a	3	3.5
1	b	1	7.0
4	b	4	7.0



Mirando aquí con atención podemos ver que los valores de índice para `left1` se han conservado, mientras que en otros ejemplos anteriores los índices de los objetos DataFrame de entrada han quedado fuera. Como el índice de `right1` es único, esta unión de «muchos a uno» (con el método predeterminado `how="inner"`) puede mantener los valores de índice de `left1` que corresponden a las filas del resultado.

Como el método de fusión predeterminado es hacer la intersección de las claves de unión, en lugar de ello se puede formar su unión con una unión externa:

```
In [65]: pd.merge(left1, right1, left_on="key",
right_index=True, how="outer")
Out[65]:
```

	key	value	group_val
0	a	0	3.5
2	a	2	3.5
3	a	3	3.5
1	b	1	7.0
4	b	4	7.0
5	c	5	NaN

Con datos jerárquicamente indexados, las cosas son más complicadas, ya que unir según el índice es equivalente a una fusión de varias claves:

```
In      left1          = ["Ohio", "Ohio", "Ohio",
[66]:    pd.DataFrame({"key1":
```

```
....:             "Nevada", "Nevada"],  
....:     "key2": [2000, 2001, 2002, 2001,  
....:             2002],  
....:     "data": pd.Series(range(5),  
....:                         dtype="Int64")})
```

```
In [67]: righth_index = pd.MultiIndex.from_arrays(  
....:         [  
....:             ["Nevada", "Nevada", "Ohio", "Ohio", "Ohio",  
....:             "Ohio"],  
....:             [2001, 2000, 2000, 2000, 2001, 2002]  
....:         ]  
....:  
)
```

```
In [68]: righth = pd.DataFrame({"event1": pd.Series([0, 2,  
4, 6, 8, 10], dtype="Int64",  
....:     index=righth_index),  
....:     "event2": pd.Series([1, 3, 5, 7, 9, 11],  
....:     dtype="Int64",  
....:     index=righth_index)})
```

```
In [69]: left  
Out[69]:
```

	key1	key2	data
0	Ohio	2000	0
1	Ohio	2001	1
2	Ohio	2002	2
3	Nevada	2001	3
4	Nevada	2002	4

```
In [70]: righth  
Out[70]:
```

	event1	event2
Nevada	2001	0 1

		2000	2	3
Ohio	2000		4	5
	2000		6	7
	2001		8	9
	2002		10	11

En este caso, hay que indicar las distintas columnas que se van a unir como una lista (como se puede observar, los valores de índice duplicados se manejan con `how="outer"`):

```
In [71]: pd.merge(left, right, left_on=["key1", "key2"],
right_index=True)
Out[71]:
```

	key1	key2	data	event1	event2
0	Ohio	2000	0	4	5
0	Ohio	2000	0	6	7
1	Ohio	2001	1	8	9
2	Ohio	2002	2	10	11
3	Nevada	2001	3	0	1

```
In [72]: pd.merge(left, right, left_on=["key1", "key2"],
.....: right_index=True, how="outer")
Out[72]:
```

	key1	key2	data	event1	event2
0	Ohio	2000	0	4	5
0	Ohio	2000	0	6	7
1	Ohio	2001	1	8	9
2	Ohio	2002	2	10	11
3	Nevada	2001	3	0	1
4	Nevada	2002	4	<NA>	<NA>
4	Nevada	2000	<NA>	2	3

También es posible utilizar los índices de ambos lados de la fusión:

```
In [73]: left2 = pd.DataFrame([[1., 2.], [3., 4.], [5.,
6.]])
```

```
....:     index=["a", "c", "e"],  
....:     columns=["Ohio", "Nevada"]).astype("Int64")
```

```
In [74]: right2 = pd.DataFrame([[7., 8.], [9., 10.], [11.,  
12.], [13, 14]]>,
```

```
....:     index=["b", "c", "d", "e"],  
....:     columns=["Missouri", "Alabama"]).
```

```
astype("Int64")
```

```
In [75]: left2  
Out[75]:
```

	Ohio	Nevada
a	1	2
c	3	4
e	5	6

```
In [76]: right2
```

```
Out[76]:
```

	Missouri	Alabama
b	7	8
c	9	10
d	11	12
e	13	14

```
In [77]: pd.merge(left2, right2, how="outer",  
left_index=True, right_index=True)
```

```
Out[77]:
```

	Ohio	Nevada	Missouri	Alabama
a	1	2	<NA>	<NA>
b	<NA>	<NA>	7	8
c	3	4	9	10
d	<NA>	<NA>	11	12
e	5	6	13	14

El objeto DataFrame tiene un método de instancia `join` para simplificar la realización de uniones según el índice. También se puede utilizar para combinar muchos objetos DataFrame que tengan los mismos índices o parecidos pero columnas que no se superponen. En el ejemplo anterior podríamos haber escrito lo siguiente:

```
In [78]: left2.join(right2, how="outer")
Out[78]:
```

	Ohio	Nevada	Missouri	Alabama
a	1	2	<NA>	<NA>
b	<NA>	<NA>	7	8
c	3	4	9	10
d	<NA>	<NA>	11	12
e	5	6	13	14

Comparado con `pandas.merge`, el método `join` del objeto DataFrame realiza de forma predeterminada una unión por la izquierda según las claves de unión. También soporta unir el índice del dataframe pasado en una de las columnas del dataframe:

```
In [79]: left1.join(right1, on="key")
Out[79]:
```

	key	value	group_val
0	a	0	3.5
1	b	1	7.0
2	a	2	3.5
3	a	3	3.5
4	b	4	7.0
5	c	5	NaN

Podemos pensar en este método como en la unión de datos «dentro» del objeto cuyo método `join` fue llamado.

Por último, para uniones sencillas índice sobre índice, se puede pasar una lista de dataframes para unir como alternativa a emplear la función

pandas.concat más general, que se describe en la siguiente sección:

```
In [80]: another = pd.DataFrame([[7., 8.], [9., 10.], [11.,
12.], [16., 17.]],
```

```
....:         index=["a", "c", "e", "f"],
....:         columns=["New York", "Oregon"])
```

```
In [81]: another
```

```
Out[81]:
```

		NewYork	Oregon
a		7.0	8.0
c		9.0	10.0
e		11.0	12.0
f		16.0	17.0

```
In [82]: left2.join([right2, another])
```

```
Out[82]:
```

	Ohio	Nevada	Missouri	Alabama	NewYork	Oregon
a	1	2	<NA>	<NA>	7.0	8.0
c	3	4	9	10	9.0	10.0
e	5	6	13	14	11.0	12.0

```
In [83]: left2.join([right2, another], how="outer")
```

```
Out[83]:
```

	Ohio	Nevada	Missouri	Alabama	NewYork	Oregon
a	1	2	<NA>	<NA>	7.0	8.0
c	3	4	9	10	9.0	10.0
e	5	6	13	14	11.0	12.0
b	<NA>	<NA>	7	8	NaN	NaN
d	<NA>	<NA>	11	12	NaN	NaN
f	<NA>	<NA>	<NA>	<NA>	16.0	17.0

Concatenación a lo largo de un eje

Existe otro tipo de operación de combinación de datos a la que se denomina de forma intercambiable concatenación o apilamiento. La función concatenate de NumPy puede hacerlo con arrays NumPy:

```
In [84]: arr = np.arange(12).reshape((3, 4))
```

```
In [85]: arr  
Out[85]:
```

```
array([ [ 0,  1,  2,  3],  
       [ 4,  5,  6,  7],  
       [ 8,  9, 10, 11]])
```

```
In [86]: np.concatenate([arr, arr], axis=1)  
Out[86]:
```

```
array([ [ 0,  1,  2,  3,  0,  1,  2,  3],  
       [ 4,  5,  6,  7,  4,  5,  6,  7],  
       [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

En el contexto de objetos pandas, como Series y DataFrame, tener ejes etiquetados permite generalizar aun más la concatenación de arrays. En este caso en particular, tenemos unos cuantos interrogantes adicionales:

- Si los objetos se indexan de manera diferente en los otros ejes, ¿debemos combinar los distintos elementos de estos ejes o utilizar solamente los valores en común?
- ¿Es necesario que los fragmentos de datos concatenados sean identificables como tales en el objeto resultante?
- ¿Contiene el «eje de concatenación» los datos que necesita para ser conservado? En muchos casos, es mejor que las etiquetas enteras predeterminadas de un dataframe no se conserven durante la concatenación.

La función concat de pandas ofrece una manera consistente de dar respuesta a cada una de estas preguntas. Daré unos cuantos ejemplos para

ilustrar su funcionamiento. Supongamos que tenemos tres series sin superposición de índice:

```
In [87]: s1 = pd.Series([0, 1], index=["a", "b"],  
dtype="Int64")
```

```
In [88]: s2 = pd.Series([2, 3, 4], index=["c", "d", "e"],  
dtype="Int64")
```

```
In [89]: s3 = pd.Series([5, 6], index=["f", "g"],  
dtype="Int64")
```

Llamar a pandas.concat con estos objetos en una lista junta los valores y los índices:

```
In [90]: s1  
Out[90]:
```

a	0
b	1
dtype:	Int64

```
In [91]: s2  
Out[91]:
```

c	2
d	3
e	4
dtype:	Int64

```
In [92]: s3  
Out[92]:
```

f	5
g	6
dtype:	Int64

```
In [93]: pd.concat([s1, s2, s3])  
Out[93]:
```

a	0
---	---

```
b          1  
c          2  
d          3  
e          4  
f          5  
g          6  
dtype: Int64
```

De forma predeterminada, `pandas.concat` trabaja sobre `axis="index"`, produciendo otra serie. Si se pasa `axis="columns"`, el resultado será sin embargo un dataframe:

```
In [94]: pd.concat([s1, s2, s3], axis="columns")  
Out[94]:
```

	0	1	2
a	0	<NA>	<NA>
b	1	<NA>	<NA>
c	<NA>	2	<NA>
d	<NA>	3	<NA>
e	<NA>	4	<NA>
f	<NA>	<NA>	5
g	<NA>	<NA>	6

En este caso no hay superposición en el otro eje, que, como se puede ver, es la combinación (la unión “outer”) de los índices. En vez de esto, se pueden cruzar pasando `join="inner"`:

```
In [95]: s4 = pd.concat([s1, s3])
```

```
In [96]: s4  
Out[96]:
```

```
a          0  
b          1  
f          5  
g          6  
dtype: Int64
```

```
In [97]: pd.concat([s1, s4], axis="columns")
Out[97]:
```

	0	1
a	0	0
b	1	1
f	<NA>	5
g	<NA>	6

```
In [98]: pd.concat([s1, s4], axis="columns", join="inner")
Out[98]:
```

	0	1
a	0	0
b	1	1

En este último ejemplo, las etiquetas “f” y “g” desaparecieron debido a la opción `join="inner"`.

Un posible problema es que las piezas concatenadas no se puedan identificar en el resultado. Supongamos entonces que queremos crear un índice jerárquico en el eje de concatenación. Para ello, empleamos el argumento `keys`:

```
In [99]: result = pd.concat([s1, s1, s3], keys=["one",
"two", "three"])
```

```
In [100]: result
Out[100]:
```

one	a	0
	b	1
two	a	0
	b	1
three	f	5
	g	6
dtype:	Int64	

```
In [101]: result.unstack()  
Out[101]:
```

	a	b	f	g
one	0	1	<NA>	<NA>
two	0	1	<NA>	<NA>
three	<NA>	<NA>	5	6

En el caso de combinar series sobre axis="columns", las keys se convierten en los encabezados de columna del dataframe:

```
In [102]: pd.concat([s1, s2, s3], axis="columns", keys=["one", "two", "three"])  
Out[102]:
```

	one	two	three
a	0	<NA>	<NA>
b	1	<NA>	<NA>
c	<NA>	2	<NA>
d	<NA>	3	<NA>
e	<NA>	4	<NA>
f	<NA>	<NA>	5
g	<NA>	<NA>	6

La misma lógica aplica a los objetos DataFrame:

```
In [103]: df1 = pd.DataFrame(np.arange(6).reshape(3, 2),  
index=["a", "b", "c"],  
.....: columns=["one", "two"])
```

```
In [104]: df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2),  
index=["a", "c"],  
.....: columns=["three", "four"])
```

```
In [105]: df1  
Out[105]:
```

	one	two
a	0	1
b	2	3
c	4	5

In [106]: df2

Out[106]:

	three	four
a	5	6
c	7	8

In [107]: pd.concat([df1, df2], axis="columns", keys=[“level1”, “level2”])

Out[107]:

	level1	level2		
	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

Aquí se utiliza el argumento keys para crear un índice jerárquico en el que se pueda utilizar el primer nivel para identificar cada uno de los objetos DataFrame concatenados.

Si se pasa un diccionario de objetos en lugar de una lista, se emplearán las claves del diccionario para la opción keys:

In [108]: pd.concat({“level1”: df1, “level2”: df2}, axis="columns")

Out[108]:

	level1	level2		
	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

Hay argumentos adicionales que gestionan el modo en que se crea el índice jerárquico (véase la tabla 8.3). Por ejemplo, podemos asignar nombre a los niveles de eje creados con el argumento `names`:

```
In [109]: pd.concat([df1, df2], axis="columns", keys=["level1", "level2"],  
.....:           names=["upper", "lower"])
```

Out[109]:

upper	level1		level2	
lower	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

Una última consideración tiene que ver con los objetos `DataFrames` en los que el índice de fila no contiene datos de importancia:

```
In [110]: df1 = pd.DataFrame(np.random.standard_normal((3,  
4)),
```

```
.....:           columns=["a", "b", "c", "d"])
```

```
In [111]: df2 = pd.DataFrame(np.random.standard_normal((2,  
3)),
```

```
.....:           columns=["b", "d", "a"])
```

```
In [112]: df1
```

Out[112]:

	a	b	c	d
0	1.248804	0.774191	-0.319657	-0.624964
1	1.078814	0.544647	0.855588	1.343268
2	-0.267175	1.793095	-0.652929	-1.886837

```
In [113]: df2  
Out[113]:
```

	b	d	a
0	1.059626	0.644448	-0.007799
1	-0.449204	2.448963	0.667226

En este caso se puede pasar `ignore_index=True`, que descarta los índices de cada dataframe y concatena los datos solo de las columnas, asignando un nuevo índice por defecto:

```
In [114]: pd.concat([df1, df2], ignore_index=True)  
Out[114]:
```

	a	b	c	d
0	1.248804	0.774191	-0.319657	-0.624964
1	1.078814	0.544647	0.855588	1.343268
2	-0.267175	1.793095	-0.652929	-1.886837
3	-0.007799	1.059626	NaN	0.644448
4	0.667226	-0.449204	NaN	2.448963

La tabla 8.3 describe los argumentos de la función `pandas.concat`.

Tabla 8.3. Argumentos de la función pandas.concat.

Argumento	Descripción
<code>objs</code>	Lista o diccionario de objetos pandas a concatenar; este es el único argumento requerido.
<code>axis</code>	Eje a lo largo del cual concatenar; por defecto concatena a lo largo de filas (<code>axis="index"</code>).
<code>join</code>	Puede ser "inner" u "outer" ("outer" por defecto); ya sea para encontrar la intersección (interno) o para unir (externo) índices a lo largo de los otros ejes.
<code>keys</code>	Valores que se asocian con los objetos que se van a concatenar, formando un índice jerárquico a lo largo de los ejes de concatenación; puede ser una lista o array de valores arbitrarios, un array de tuplas o una lista de arrays (si se pasan arrays de varios niveles en <code>levels</code>).

levels	Índices específicos que se utilizan como índice jerárquico si se pasa keys y/o levels.
names	Nombres para niveles jerárquicos creados si se pasa keys y/o levels.
verify_integrity	Busca duplicados en el nuevo eje del objeto concatenado y da un error si los hay; de forma predeterminada (False) permite duplicados.
ignore_index	No conserva los índices a lo largo del axis de concatenación, sino que produce un nuevo índice range(total_index).

Combinar datos con superposición

Hay otra situación de combinación de datos que no se puede expresar como una operación de fusión o concatenación. Es posible que tengamos dos conjuntos de datos con índices que se superponen totalmente o en parte. Como ejemplo motivador, veamos la función where de NumPy, que realiza el equivalente orientado a arrays de una expresión if-else:

```
In [115]: a = pd.Series([np.nan, 2.5, 0.0, 3.5, 4.5,
np.nan],
```

```
.....: index=["f", "e", "d", "c", "b", "a"])
```

```
In [116]: b = pd.Series([0., np.nan, 2., np.nan, np.nan,
5.],
```

```
.....: index=["a", "b", "c", "d", "e", "f"])
```

```
In [117]: a
Out[117]:
```

f	NaN
e	2.5
d	0.0
c	3.5
b	4.5
a	NaN
dtype:	float64

```
In [118]: b
Out[118]:
```

a	0.0
b	NaN
c	2.0
d	NaN
e	NaN
f	5.0
dtype:	float64

```
In [119]: np.where(pd.isna(a), b, a)
Out[119]: array([0., 2.5, 0., 3.5, 4.5, 5.])
```

Aquí, siempre que los valores de a son nulos, se seleccionan valores de b, en otro caso se seleccionan los valores no nulos de a. Utilizar numpy.where no comprueba si las etiquetas de índice están alineadas o no (y ni siquiera requiere que los objetos tengan la misma longitud), de modo que si se desea alinear valores por índice, es mejor utilizar el método `combine_first` del objeto Series:

```
In [120]: a.combine_first(b)
Out[120]:
```

a	0.0
b	4.5
c	3.5
d	0.0
e	2.5
f	5.0
dtype:	float64

Con objetos DataFrame, `combine_first` hace lo mismo columna a columna, de forma que se puede pensar que este método lo que hace es «parchear» los datos que faltan en el objeto de llamada con datos del objeto que se pase:

```
In [121]: df1 = pd.DataFrame({"a": [1., np.nan, 5., np.nan],
```

```
....:         "b": [np.nan, 2., np.nan, 6.],  
....:         "c": range(2, 18, 4)})
```

```
In [122]: df2 = pd.DataFrame({"a": [5., 4., np.nan, 3., 7.],  
....:         "b": [np.nan, 3., 4., 6., 8.]})
```

```
In [123]: df1  
Out[123]:
```

	a	b	c
0	1.0	NaN	2
1	NaN	2.0	6
2	5.0	NaN	10
3	NaN	6.0	14

```
In [124]: df2  
Out[124]:
```

	a	b
0	5.0	NaN
1	4.0	3.0
2	NaN	4.0
3	3.0	6.0
4	7.0	8.0

```
In [125]: df1.combine_first(df2)  
Out[125]:
```

	a	b	c
0	1.0	NaN	2.0
1	4.0	2.0	6.0
2	5.0	4.0	10.0
3	3.0	6.0	14.0
4	7.0	8.0	NaN

El resultado de `combine_first` con objetos DataFrame tendrá la unión de todos los nombres de columna.

8.3 Remodelación y transposición

Para reordenar datos tabulares existen una serie de operaciones básicas, denominadas operaciones de remodelación o transposición.

Remodelación con indexación jerárquica

La indexación jerárquica ofrece una forma coherente de reordenar datos en un dataframe. Tenemos dos acciones principales:

- `stack`: Cambia o transpone las columnas de los datos por las filas.
- `unstack`: Cambia o transpone las filas por las columnas.

Ilustraré estas operaciones mediante una serie de ejemplos. Supongamos un pequeño dataframe con arrays de cadena de texto como índices de fila y columna:

```
In [126]: data = pd.DataFrame(np.arange(6).reshape((2, 3)),  
.....:     index=pd.Index(["Ohio", "Colorado"],  
.....:     name="state"),  
.....:     columns=pd.Index(["one", "two", "three"],  
.....:     name="number"))
```

```
In [127]: data  
Out[127]:
```

	one	two	three
number			
state			
Ohio	0	1	2
Colorado	3	4	5

Utilizando el método `stack` con estos datos cambiamos las columnas por las filas, produciendo una serie:

```
In [128]: result = data.stack()
```

```
In [129]: result  
Out[129]:
```

```
state          number  
Ohio           one      0  
                 two      1  
                 three    2  
Colorado       one      3  
                 two      4  
                 three    5  
  
dtype: int64
```

A partir de una serie indexada jerárquicamente, se pueden reordenar los datos de nuevo en un dataframe con `unstack`:

```
In [130]: result.unstack()  
Out[130]:
```

```
number          one      two      three  
state  
Ohio            0        1        2  
Colorado        3        4        5
```

De forma predeterminada, el nivel más interno no está apilado (igual que con `stack`). Se puede desapilar otro nivel pasando un número de nivel o un nombre:

```
In [131]: result.unstack(level=0)  
Out[131]:
```

```
state          Ohio          Colorado  
number  
one            0            3  
two            1            4  
three          2            5
```

```
In [132]: result.unstack(level="state")
Out[132]:
```

	Ohio	Colorado
state		
number		
one	0	3
two	1	4
three	2	5

Desapilar podría introducir datos ausentes si no se encuentran todos los valores del nivel en cada subgrupo:

```
In [133]: s1 = pd.Series([0, 1, 2, 3], index=["a", "b", "c", "d"], dtype="Int64")
```

```
In [134]: s2 = pd.Series([4, 5, 6], index=["c", "d", "e"], dtype="Int64")
```

```
In [135]: data2 = pd.concat([s1, s2], keys=["one", "two"])
```

```
In [136]: data2
```

```
Out[136]:
```

one	a	0
	b	1
	c	2
	d	3
two	c	4
	d	5
	e	6

```
dtype: Int64
```

Apilar filtra los datos ausentes de forma predeterminada, de modo que la operación se puede invertir con mayor facilidad:

```
In [137]: data2.unstack()
Out[137]:
```

```
          a      b      c      d      e
one      0      1      2      3    <NA>
two    <NA>  <NA>    4      5      6
```

```
In [138]: data2.unstack().stack()
Out[138]:
```

```
one      a      0
        b      1
        c      2
        d      3
two      c      4
        d      5
        e      6
dtype: Int64
```

```
In [139]: data2.unstack().stack(dropna=False)
Out[139]:
```

```
one      a      0
        b      1
        c      2
        d      3
        e    <NA>
two      a    <NA>
        b    <NA>
        c      4
        d      5
        e      6
dtype: Int64
```

Cuando se desapila en un dataframe, el nivel desapilado se convierte en el nivel inferior del resultado:

```
In [140]: df = pd.DataFrame({"left": result, "right": result
+ 5},
.....:     columns=pd.Index(["left", "right"],
```

```
name="side"))
```

```
In [141]: df  
Out[141]:
```

			left	right
side				
state		number		
Ohio		one	0	5
		two	1	6
		three	2	7
Colorado		one	3	8
		two	4	9
		three	5	10

```
In [142]: df.unstack(level="state")  
Out[142]:
```

side	left		right	
state	Ohio	Colorado	Ohio	Colorado
number				
one	0	3	5	8
two	1	4	6	9
three	2	5	7	10

Como con unstack, al llamar a stack podemos indicar el nombre del eje que queremos apilar:

```
In [143]: df.unstack(level="state").stack(level="side")  
Out[143]:
```

state	Colorado	Ohio
number	side	
one	left	3
	right	8
two	left	4
	right	9

three	left	5	2
	right	10	7

Transponer del formato «largo» al «ancho»

Una forma habitual de almacenar varias series temporales en bases de datos y archivos CSV es el llamado formato largo o apilado. En este formato, los valores individuales se representan con una sola fila de una tabla en lugar de utilizar varios valores por fila.

Carguemos algunos datos de ejemplo y hagamos un poco de disputa de series temporales y otras limpiezas de datos:

```
In [144]: data = pd.read_csv("examples/macrodata.csv")
In [145]: data = data.loc[:, ["year", "quarter", "realgdp",
                           "infl", "unemp"]]
In [146]: data.head()
Out[146]:
```

	year	quarter	realgdp	infl	unemp
0	1959	1	2710.349	0.00	5.8
1	1959	2	2778.801	2.34	5.1
2	1959	3	2775.488	2.74	5.3
3	1959	4	2785.204	0.27	5.6
4	1960	1	2847.699	2.31	5.2

Primero, uso pandas.PeriodIndex (que representa intervalos de tiempo en lugar de puntos en el tiempo, y de la que hablaremos con más detalle en el capítulo 11), para combinar las columnas year y quarter y fijar así el índice para que esté formado por valores datetime al final de cada trimestre:

```
In [147]: periods = pd.PeriodIndex(year=data.pop("year"),
.....:                     quarter=data.pop("quarter"),
.....:                     name="date")
```

```
In [148]: periods
```

```
Out[148]:
```

```
PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4',
              '1960Q1', '1960Q2',
              '1960Q3', '1960Q4', '1961Q1', '1961Q2',
              ...
              '2007Q2', '2007Q3', '2007Q4', '2008Q1',
              '2008Q2', '2008Q3',
              '2008Q4', '2009Q1', '2009Q2', '2009Q3'],
              dtype='period[Q-DEC]', name='date',
              length=203)
```

```
In [149]: data.index = periods.to_timestamp("D")
```

```
In [150]: data.head()
```

```
Out[150]:
```

	realgdp	infl	unemp
date			
1959-01-01	2710.349	0.00	5.8
1959-04-01	2778.801	2.34	5.1
1959-07-01	2775.488	2.74	5.3
1959-10-01	2785.204	0.27	5.6
1960-01-01	2847.699	2.31	5.2

Aquí he utilizado el método pop en el dataframe, que devuelve una columna mientras la borra del dataframe al mismo tiempo.

Después, selecciono un subconjunto de columnas y le doy al índice de las columnas el nombre “item”:

```
In [151]: data = data.reindex(columns=["realgdp", "infl",
                                         "unemp"])
```

```
In [152]: data.columns.name = "item"
```

```
In [153]: data.head()
```

```
Out[153]:
```

item	realgdp	infl	unemp
date			
1959-01-01	2710.349	0.00	5.8
1959-04-01	2778.801	2.34	5.1
1959-07-01	2775.488	2.74	5.3
1959-10-01	2785.204	0.27	5.6
1960-01-01	2847.699	2.31	5.2

Por último, remodelo con stack, convierto los nuevos niveles de índice en columnas con reset_index, y doy finalmente a la columna que contiene los valores de datos el nombre “value”:

```
In [154]: long_data = (data.stack()
.....
.....
.....: .reset_index()
.....: .rename(columns={0: "value"}))
```

Ahora, ldata tiene este aspecto:

```
In [155]: long_data[:10]
Out[155]:
```

	date	item	value
0	1959-01-01	realgdp	2710.349
1	1959-01-01	infl	0.000
2	1959-01-01	unemp	5.800
3	1959-04-01	realgdp	2778.801
4	1959-04-01	infl	2.340
5	1959-04-01	unemp	5.100
6	1959-07-01	realgdp	2775.488
7	1959-07-01	infl	2.740
8	1959-07-01	unemp	5.300
9	1959-10-01	realgdp	2785.204

En este formato denominado largo para varias series temporales, cada fila de la tabla representa una sola observación.

Con frecuencia, los datos se almacenan de esta forma en bases de datos SQL relacionales, porque un esquema fijo (nombres de columna y tipos de datos) permite que el número de valores distintos de la columna `item` cambie a medida que se añaden datos a la tabla. En el ejemplo anterior, `date` y `item` serían normalmente las claves principales (en jerga de bases de datos relacionales), que ofrecen integridad relacional y uniones más sencillas.

En algunos casos, puede ser más difícil trabajar con los datos con este formato; quizás es preferible tener un dataframe que contenga una columna por valor `item` diferente indexada por marcas de tiempo en la columna `date`. El método `pivot` del objeto DataFrame realiza exactamente esta transformación:

```
In [156]: pivoted = long_data.pivot(index="date",
columns="item",
.....:           values="value")
```

```
In [157]: pivoted.head()
Out[157]:
```

item	infl	realgdp	unemp
date			
1959-01-01	0.00	2710.349	5.8
1959-04-01	2.34	2778.801	5.1
1959-07-01	2.74	2775.488	5.3
1959-10-01	0.27	2785.204	5.6
1960-01-01	2.31	2847.699	5.2

Los primeros dos valores pasados son las columnas que se van a utilizar, respectivamente, como índice de fila y columna, y después finalmente una columna de valor opcional para llenar el dataframe. Supongamos que tenemos dos columnas de valor que queremos remodelar al mismo tiempo:

```
In [158]: long_data["value2"] =
np.random.standard_normal(len(long_data))
```

```
In [159]: long_data[:10]
Out[159]:
```

	date	item	value	value2
0	1959-01-01	realgdp	2710.349	0.802926
1	1959-01-01	infl	0.000	0.575721
2	1959-01-01	unemp	5.800	1.381918
3	1959-04-01	realgdp	2778.801	0.000992
4	1959-04-01	infl	2.340	-0.143492
5	1959-04-01	unemp	5.100	-0.206282
6	1959-07-01	realgdp	2775.488	-0.222392
7	1959-07-01	infl	2.740	-1.682403
8	1959-07-01	unemp	5.300	1.811659
9	1959-10-01	realgdp	2785.204	-0.351305

Omitiendo el último argumento, se obtiene un dataframe con columnas jerárquicas:

```
In [160]: pivoted = long_data.pivot(index="date",
columns="item")
```

```
In [161]: pivoted.head()
Out[161]:
```

	value			value2		
item	infl	realgdp	unemp	infl	realgdp	unemp
date						
1959-01-01	0.00	2710.349	5.8	0.575721	0.802926	1.381918
1959-04-01	2.34	2778.801	5.1	-0.143492	0.000992	-0.206282
1959-07-01	2.74	2775.488	5.3	-1.682403	-0.222392	1.811659
1959-10-01	0.27	2785.204	5.6	0.128317	-0.351305	-1.313554
1960-01-01	2.31	2847.699	5.2	-0.615939	0.498327	0.174072

```
In [162]: pivoted["value"].head()
Out[162]:
```

item	infl	realgdp	unemp
date			
1959-01-01			5.8

	0.00	2710.349	
1959-04-01	2.34	2778.801	5.1
1959-07-01	2.74	2775.488	5.3
1959-10-01	0.27	2785.204	5.6
1960-01-01	2.31	2847.699	5.2

Hay que tener en cuenta que pivot es equivalente a crear un índice jerárquico utilizando set_index seguido de una llamada a unstack:

```
In [163]: unstacked = long_data.set_index(["date", "item"]).unstack(level="item")
```

```
In [164]: unstacked.head()
Out[164]:
```

item	value			value2		
	infl	realgdp	unemp	infl	realgdp	unemp
	date					
1959-01-01	0.00	2710.349	5.8	0.575721	0.802926	1.381918
1959-04-01	2.34	2778.801	5.1	-0.143492	0.000992	-0.206282
1959-07-01	2.74	2775.488	5.3	-1.682403	-0.222392	1.811659
1959-10-01	0.27	2785.204	5.6	0.128317	-0.351305	-1.313554
1960-01-01	2.31	2847.699	5.2	-0.615939	0.498327	0.174072

Transponer del formato «ancho» al «largo»

Una operación inversa a pivot para objetos DataFrames es pandas.melt. En lugar de transformar una columna en muchas en un nuevo dataframe, combina varias columnas en una sola, produciendo un dataframe que es más largo que la entrada. Veamos un ejemplo:

```
In [166]: df = pd.DataFrame({"key": ["foo", "bar", "baz"],
....:                   "A": [1, 2, 3],
....:                   "B": [4, 5, 6],
....:                   "C": [7, 8, 9]})
```

```
In [167]: df
```

```
Out[167]:
```

	key	A	B	C
0	foo	1	4	7
1	bar	2	5	8
2	baz	3	6	9

La columna “key” puede ser un indicador de grupo, y las otras columnas son valores de datos. Cuando utilizamos pandas.melt, debemos indicar qué columnas (si hay alguna) son indicadores de grupo. Vamos a usar aquí “key” como único indicador de grupo:

```
In [168]: melted = pd.melt(df, id_vars="key")
```

```
In [169]: melted
```

```
Out[169]:
```

	key	variable	value
0	foo	A	1
1	bar	A	2
2	baz	A	3
3	foo	B	4
4	bar	B	5
5	baz	B	6
6	foo	C	7
7	bar	C	8
8	baz	C	9

Utilizando pivot, podemos remodelar de nuevo a la disposición original:

```
In [170]: reshaped      =      melted.pivot(index="key",
columns="variable",
.....: values="value")
```

```
In [171]: reshaped
```

```
Out[171]:
```

variable	A	B	C
key			
bar	2	5	8
baz	3	6	9
foo	1	4	7

Como el resultado de pivot crea un índice a partir de la columna utilizada como etiquetas de fila, quizá nos interese utilizar reset_index para volver a colocar los datos en una columna:

```
In [172]: reshaped.reset_index()
Out[172]:
```

variable	key	A	B	C
0	bar	2	5	8
1	baz	3	6	9
2	foo	1	4	7

También se puede especificar un subconjunto de columnas que se utilizarán como columnas value:

```
In [173]: pd.melt(df, id_vars="key", value_vars=["A", "B"])
Out[173]:
```

	key	variable	value
0	foo	A	1
1	bar	A	2
2	baz	A	3
3	foo	B	4
4	bar	B	5
5	baz	B	6

pandas.melt se puede utilizar también sin identificadores de grupo:

```
In [174]: pd.melt(df, value_vars=["A", "B", "C"])
```

```
Out[174]:
```

	variable	value
0	A	1
1	A	2
2	A	3
3	B	4
4	B	5
5	B	6
6	C	7
7	C	8
8	C	9

```
In [175]: pd.melt(df, value_vars=["key", "A", "B"])
Out[175]:
```

	variable	value
0	key	foo
1	key	bar
2	key	baz
3	A	1
4	A	2
5	A	3
6	B	4
7	B	5
8	B	6

8.4 Conclusión

Ahora que mis lectores tienen ya en su haber algunos fundamentos de pandas para la importación, limpieza y reorganización de los datos, ya están preparados para pasar a su visualización con matplotlib. Volveremos a explorar otras áreas de pandas más adelante en el libro, cuando hablemos sobre análisis más avanzados.

Gráficos y visualización

Realizar visualizaciones informativas (denominadas gráficos) es una de las tareas más importantes en análisis de datos. Puede ser parte del proceso de exploración, por ejemplo, para identificar con facilidad valores atípicos o transformaciones de datos necesarias, o como una manera de generar ideas para modelos. En otros casos, quizá el objetivo final sea crear una visualización interactiva para la web. Python tiene muchas librerías adicionales para realizar visualizaciones estáticas o dinámicas, pero me voy a centrar principalmente en matplotlib (<https://matplotlib.org>) y en las librerías basadas en ella.

Este paquete de trazado de sobremesa denominado matplotlib está diseñado para crear gráficos y figuras aptas para su publicación. El proyecto fue iniciado por John Hunter en 2002 para poder disponer de una interfaz de trazado de tipo MATLAB en Python. Las comunidades de matplotlib e IPython han colaborado para simplificar la creación de gráficos interactivos a partir del shell de IPython (y ahora, Jupyter notebook). En todos los sistemas operativos, matplotlib soporta varios servidores de interfaz gráfica de usuario o GUI (*Graphical User Interface*) y puede exportar visualizaciones a todos los formatos gráficos vectoriales y rasterizados más comunes (PDF, SVG, JPG, PNG, BMP, GIF, etc.). Con excepción de algunos diagramas, casi todos los gráficos de este libro se han producido utilizando matplotlib.

Con el tiempo, matplotlib ha dado lugar a una serie de juegos de herramientas adicionales para visualización de datos que utilizan matplotlib para sus gráficos subyacentes. Una de ellas es seaborn (<http://seaborn.pydata.org>), que exploraremos más tarde en este capítulo.

La forma más sencilla de seguir los códigos de ejemplo del capítulo es obtener los gráficos en Jupyter Notebook. Para configurar esto, basta con ejecutar la siguiente sentencia en un Jupyter notebook:

```
%matplotlib inline
```

💡 Desde la primera edición de 2012 de este libro, se han creado muchas librerías de visualización de datos nuevas, algunas de las cuales (como Bokeh y Altair) aprovechan la tecnología web moderna para crear visualizaciones interactivas que se integran bien con Jupyter Notebook. En lugar de utilizar varias herramientas de visualización en este libro, he decidido quedarme con matplotlib para enseñar los fundamentos, en particular porque pandas tiene una buena integración con matplotlib. Se pueden adaptar los principios de este capítulo para aprender cómo utilizar también otras librerías de visualización.

9.1 Una breve introducción a la API matplotlib

Con matplotlib, utilizamos el siguiente convenio de importación:

```
In [13]: import matplotlib.pyplot as plt
```

Después de ejecutar %matplotlib notebook en Jupyter (o simplemente %matplotlib en IPython), podemos intentar crear un gráfico sencillo. Si todo se ha configurado correctamente, debería aparecer un gráfico de líneas como el de la figura 9.1:

```
In [14]: data = np.arange(10)
```

```
In [15]: data
```

```
Out[15]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [16]: plt.plot(data)
```

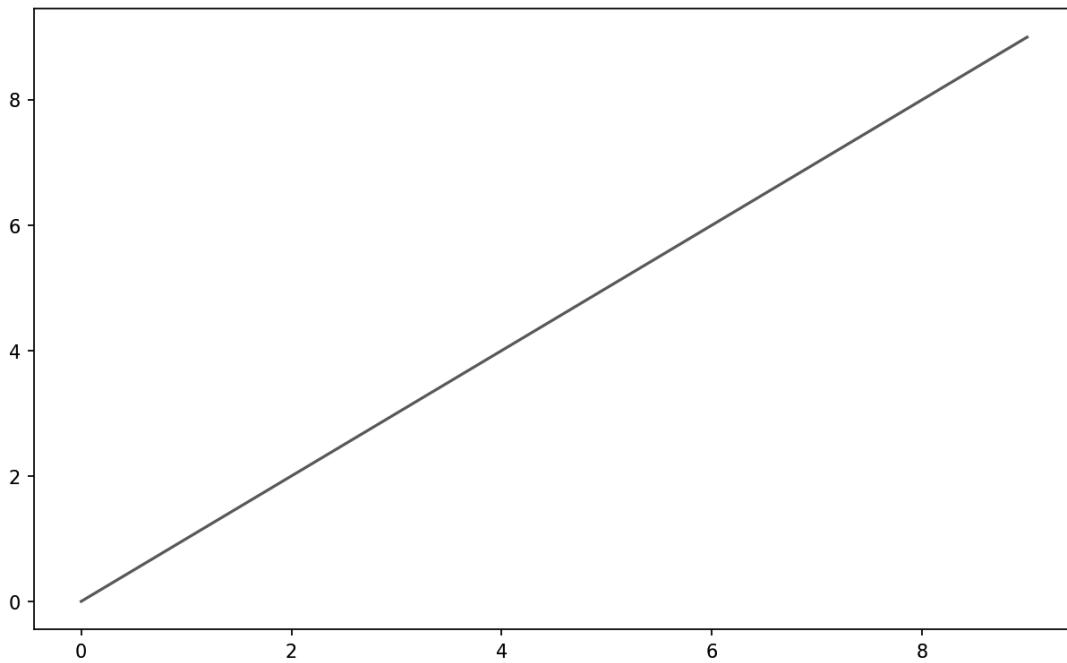


Figura 9.1. Gráfico de líneas sencillo.

Aunque librerías como seaborn y las funciones de creación de gráficos internas de pandas se encargarán de muchos de los mundanos detalles de la creación de gráficos, si se desean personalizar más allá de las opciones de función ofrecidas, será necesario aprender un poco sobre la API matplotlib.



No hay espacio suficiente en el libro para explicar con detalle la amplitud y profundidad de matplotlib. La información incluida debería bastar para empezar. La galería y la documentación de matplotlib son los mejores recursos para aprender sobre funciones avanzadas.

Figuras y subgráficos

Los gráficos en matplotlib residen dentro de un objeto `Figure`. Se puede crear una nueva figura con `plt.figure()`:

```
In [17]: fig = plt.figure()
```

En IPython, si se ejecuta primero `%matplotlib` para configurar la integración de matplotlib, aparecerá una ventana de gráfico vacía, pero en Jupyter no se verá nada hasta utilizar algunos comandos más.

La función `plt.figure` tiene distintas opciones; en particular, `figsize` garantiza que la figura tenga un determinado tamaño y una cierta proporción de aspecto si se guarda en disco. No se puede hacer un gráfico con una figura vacía. Es necesario crear uno o varios subgráficos empleando `add_subplot`:

```
In [18]: ax1 = fig.add_subplot(2, 2, 1)
```

Esto significa que la figura debería ser de 2×2 (es decir, de hasta cuatro gráficos en total), y estamos seleccionando el primero de cuatro subgráficos (numerados desde el 1). Si creamos los siguientes dos subgráficos, terminaremos con una visualización que se parece a la figura 9.2:

```
In [19]: ax2 = fig.add_subplot(2, 2, 2)
```

```
In [20]: ax3 = fig.add_subplot(2, 2, 3)
```

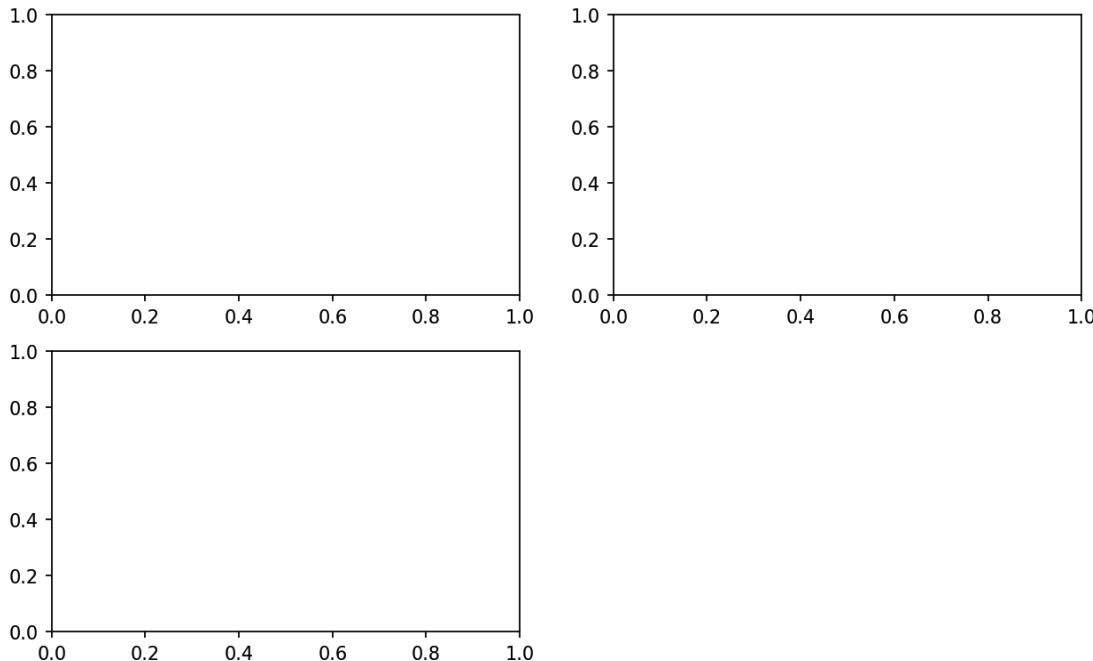


Figura 9.2. Una figura matplotlib vacía con tres subgráficos.



Un matiz del uso de notebooks de Jupyter es que los gráficos se reinicializan después de evaluarse cada celda, de modo que se deben poner todos los comandos de trazado en una sola celda del notebook.

Aquí ejecutamos todos estos comandos en la misma celda:

```
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)

ax3 = fig.add_subplot(2, 2, 3)
```

Estos objetos de eje de gráfico tienen distintos métodos que crean diferentes tipos de gráficos, así que es preferible utilizar métodos de eje en lugar de funciones de trazado de máximo nivel, como `plt.plot`. Por ejemplo, podríamos crear un gráfico de líneas con el método `plot` (véase la figura 9.3):

```
In [21]: ax3.plot(np.random.standard_normal(50).cumsum(),
color="black",
....: linestyle="dashed")
```

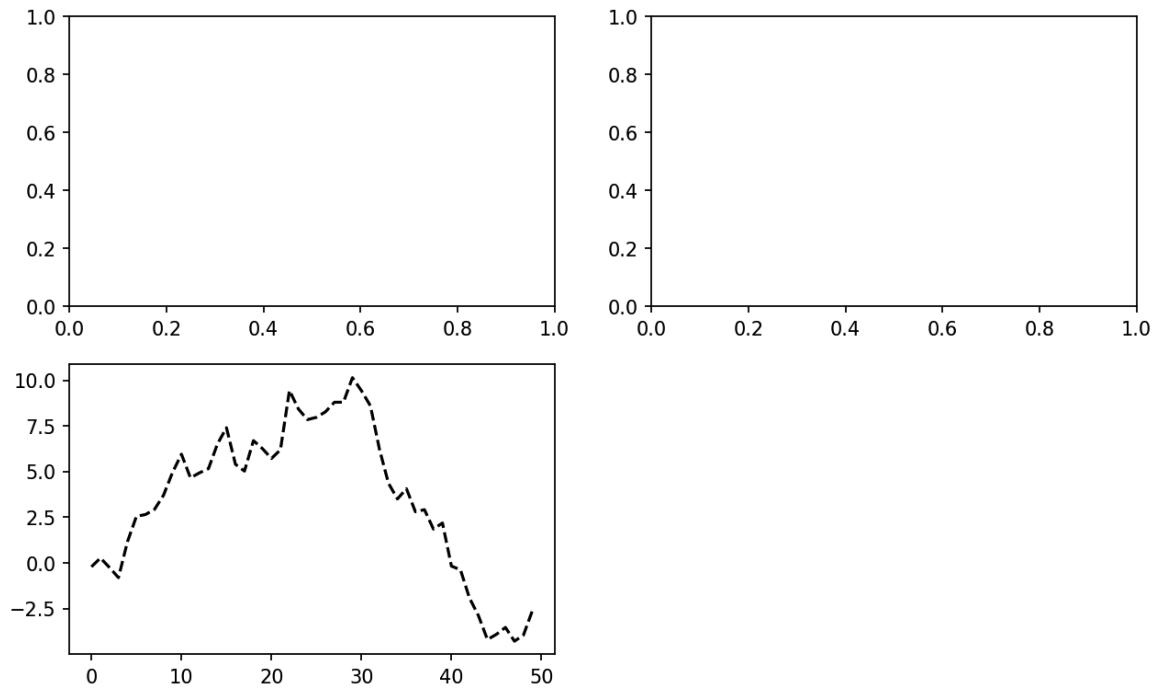


Figura 9.3. Visualización de los datos tras un solo gráfico.

Es posible observar un resultado como `<matplotlib.lines.Line2D at ...>` al ejecutar esto. `matplotlib` devuelve objetos que hacen referencia al

subcomponente del gráfico que justo se añadió. Buena parte del tiempo se puede ignorar este resultado, o bien se puede poner un punto y coma al final de la línea para suprimirlo.

Las opciones adicionales instruyen a matplotlib para que dibuje una línea negra discontinua. Los objetos devueltos por `fig.add_subplot` son aquí objetos `AxesSubplot`, en los que se puede crear directamente el gráfico en los otros subgráficos vacíos llamando al método de instancia de cada uno (figura 9.4):

```
In [22]: ax1.hist(np.random.standard_normal(100), bins=20, color="black", alpha=0.3);
```

```
In [23]: ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.standard_normal(30));
```

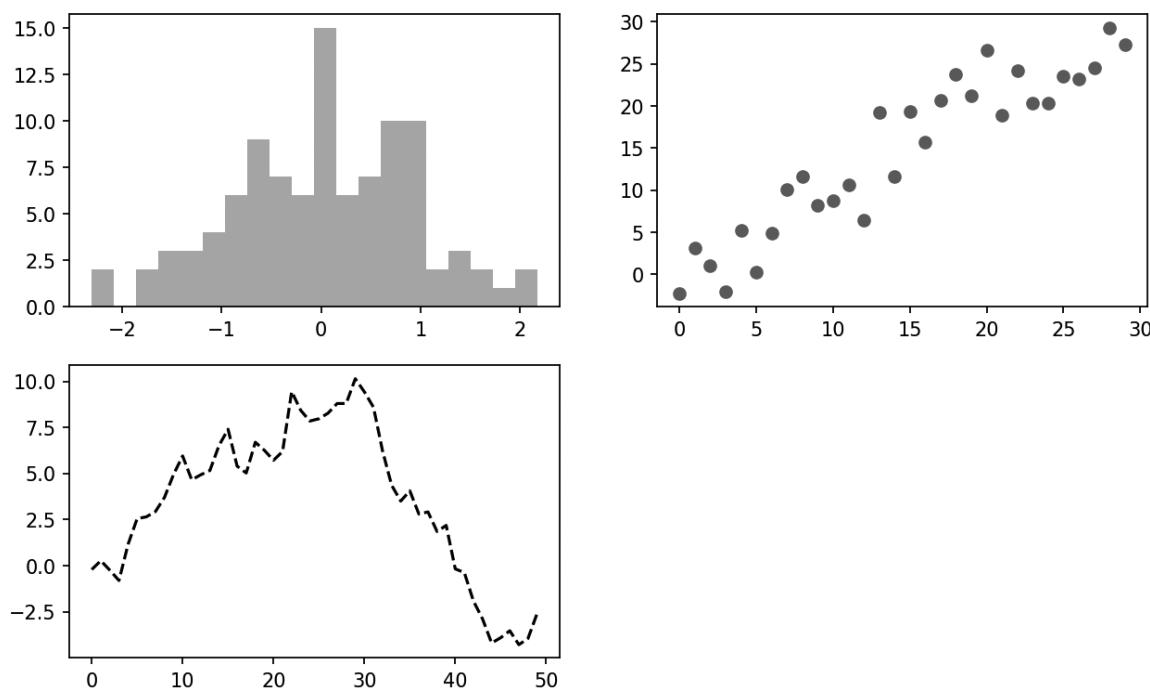


Figura 9.4. Visualización de los datos tras gráficos adicionales.

La opción de estilo `alpha=0.3` configura la transparencia del gráfico superpuesto. En la documentación de matplotlib se puede encontrar un amplio catálogo de tipos de gráficos (<https://matplotlib.org>).

Para que la creación de una cuadrícula de subgráficos sea más cómoda, matplotlib incluye un método `plt.subplots` que crea una figura nueva y devuelve un array NumPy, que contiene los objetos de subgráficos creados:

```
In [25]: fig, axes = plt.subplots(2, 3)
```

```
In [26]: axes
```

```
Out[26]:
```

```
array([<AxesSubplot:>, <AxesSubplot:>,
       <AxesSubplot:>, <AxesSubplot:>,
       <AxesSubplot:>, <AxesSubplot:>], dtype=object)
```

El array de ejes se puede indexar a continuación como un array bidimensional; por ejemplo, `axes[0, 1]` se refiere al subgráfico de la fila superior central. También se puede indicar que los subgráficos deben tener el mismo eje x o y utilizando `sharex` y `sharey`, respectivamente. Esto puede resultar muy útil cuando se comparan datos en la misma escala; de otro modo, matplotlib dimensiona automáticamente los límites de gráfico de forma independiente. Véase en la tabla 9.1 más información sobre este método.

Tabla 9.1. Opciones de `matplotlib.pyplot.subplots`.

Argumento	Descripción
<code>nrows</code>	Número de filas de subgráficos.
<code>ncols</code>	Número de columnas de subgráficos.
<code>sharex</code>	Todos los subgráficos deberían utilizar las mismas marcas del eje x (ajustar el <code>xlim</code> afectará a todos los subgráficos).
<code>sharey</code>	Todos los subgráficos deberían utilizar las mismas marcas del eje y (ajustar el <code>ylim</code> afectará a todos los subgráficos).
<code>subplot_kw</code>	Diccionario de palabras clave pasadas a la llamada <code>add_subplot</code> empleada para crear cada subgráfico.

**fig_kw	Se utilizan palabras clave adicionales a subplots al crear la figura, como plt.subplots(2, 2, figsize=(8, 6)).
----------	----------------------------------------------------------------------------------------------------------------

Ajustar el espacio alrededor de los subtrazados

De forma predeterminada, matplotlib deja una cierta cantidad de espacio libre en torno al exterior de los subgráficos y en el espaciado entre ellos. Este espacio se especifica todo en relación a la altura y anchura del gráfico, de modo que si se redimensiona el gráfico mediante código o manualmente empleando la ventana GUI, el gráfico se ajustará solo dinámicamente. Se puede cambiar el espaciado usando el método `subplots_adjust` en objetos `Figure`:

```
subplots_adjust(left=None,           bottom=None,           right=None,
                top=None,             wspace=None,            hspace=None)
```

`wspace` y `hspace` controlan el porcentaje de la anchura y altura de la figura, respectivamente, para utilizarlo como espaciado entre subgráficos. Aquí tenemos un pequeño ejemplo que se puede ejecutar en Jupyter, donde encojo el espaciado hasta cero (véase la figura 9.5):

```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
for i in range(2):
    for j in range(2):
        axes[i, j].hist(np.random.standard_normal(500), bins=50,
                        color="black", alpha=0.5)
fig.subplots_adjust(wspace=0, hspace=0)
```

Es fácil darse cuenta de que las etiquetas de los ejes se superponen. La herramienta matplotlib no comprueba esto, de modo que en un caso como este sería necesario arreglar las etiquetas a mano especificando las ubicaciones de las marcas y de las etiquetas de manera específica (veremos

cómo hacer esto en la sección «Marcas, etiquetas y leyendas» más adelante en este capítulo).

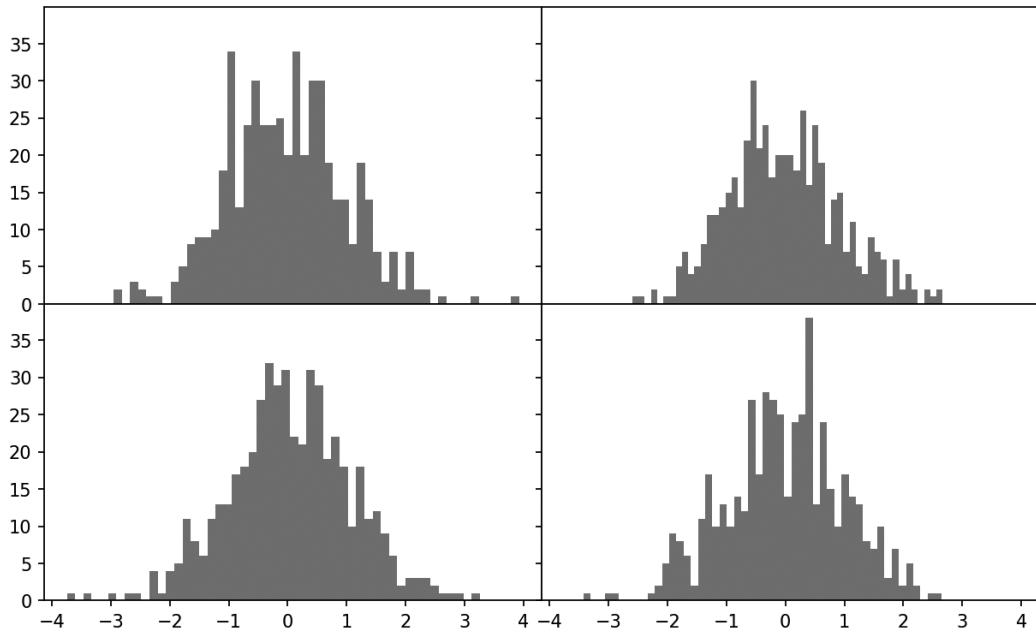


Figura 9.5. Visualización de datos sin espacio entre subgráficos.

Colores, marcadores y estilos de línea

La función de línea `plot` de matplotlib acepta arrays de coordenadas `x` e `y` y opciones de estilo de color. Por ejemplo, para dibujar `x` frente a `y` con guiones verdes, ejecutaríamos lo siguiente:

```
ax.plot(x, y, linestyle="--", color="green")
```

Se dispone de unos cuantos nombres de colores para los más utilizados, pero se puede usar cualquier color del espectro especificando su código hexadecimal (por ejemplo, “#CECECE”). El docstring de `plt.plot` incluye algunos de los estilos de línea soportados (para ello hay que ejecutar `plt.plot?` en IPython o Jupyter). La documentación en línea ofrece una referencia más amplia.

Los gráficos de líneas pueden tener de manera adicional marcadores para resaltar los puntos de datos reales. Como la función `plot` de matplotlib crea

un gráfico de líneas continuo, interpolando entre puntos, a veces puede no estar claro dónde están los mismos. El marcador se puede incluir como opción de estilo adicional (figura 9.6):

```
In [31]: ax = fig.add_subplot()  
In [32]: ax.plot(np.random.standard_normal(30).cumsum(),  
color="black",  
....:      linestyle="dashed", marker="o");
```

Para los gráficos de líneas, se puede observar que los puntos siguientes están interpolados linealmente de forma predeterminada, lo que se puede modificar con la opción `drawstyle` (véase la figura 9.7):

```
In [34]: fig = plt.figure()  
In [35]: ax = fig.add_subplot()  
In [36]: data = np.random.standard_normal(30).cumsum()  
In [37]: ax.plot(data, color="black", linestyle="dashed",  
label="Default");  
In [38]: ax.plot(data, color="black", linestyle="dashed",  
....: drawstyle="steps-post", label="steps-post");  
In [39]: ax.legend()
```

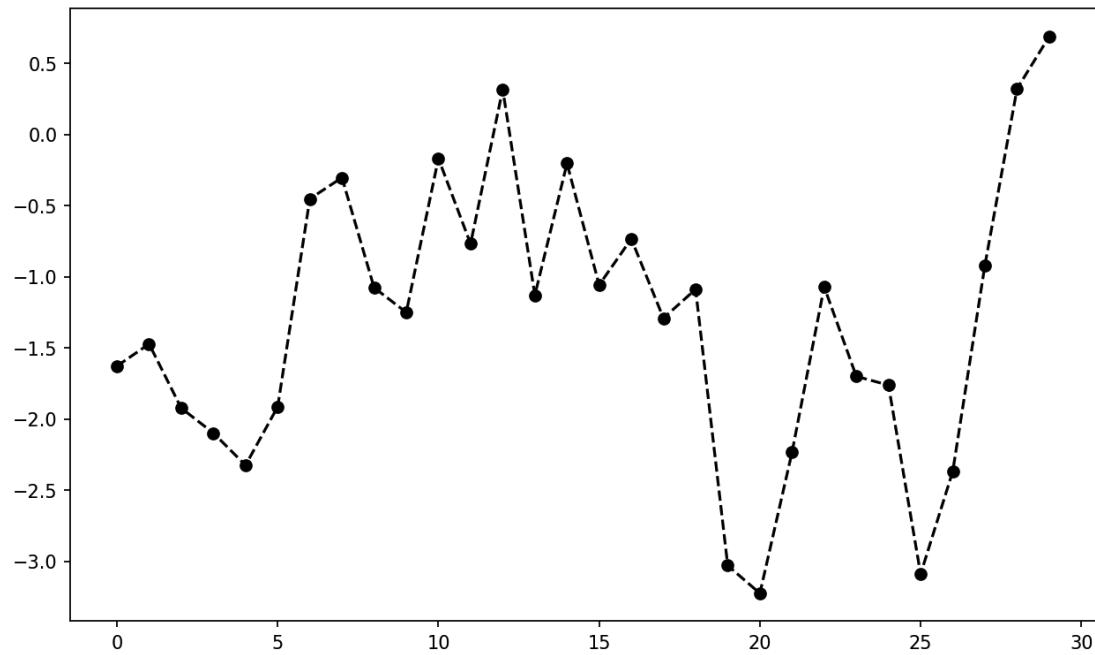


Figura 9.6. Gráfico de líneas con marcadores.

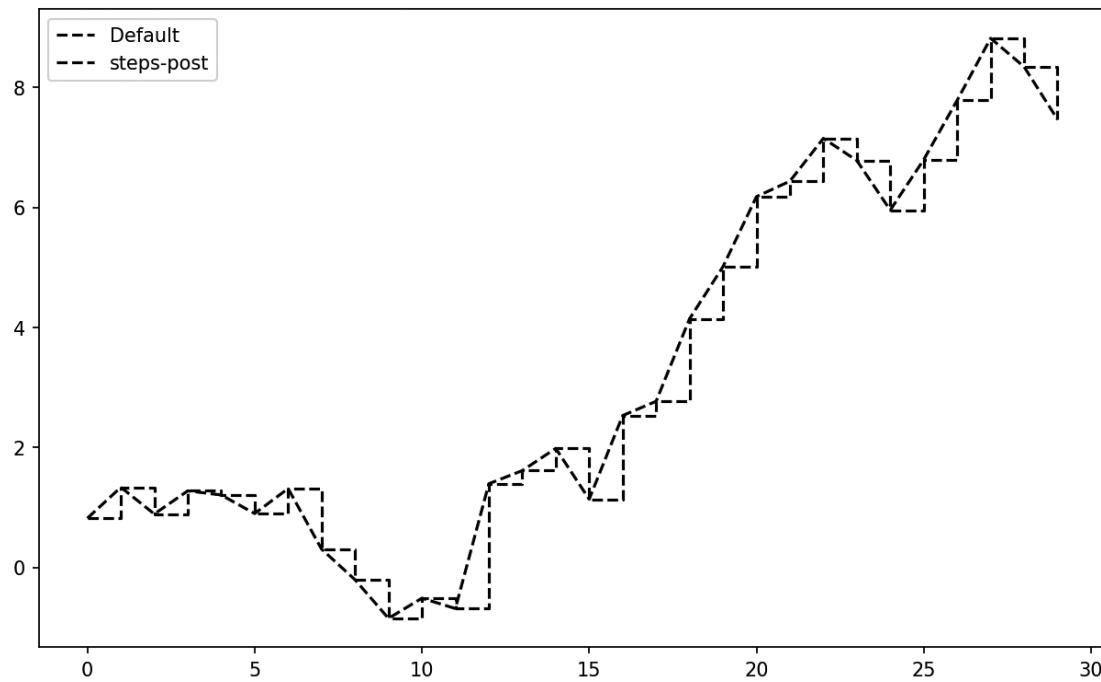


Figura 9.7. Gráfico de líneas con distintas opciones de estilo de dibujo.

Aquí, como pasamos los argumentos `label` a `plot`, podemos crear una leyenda de gráfico para identificar cada línea usando `ax.legend`. Hablo más

de las leyendas en la siguiente sección «Marcas, etiquetas y leyendas».



Para crear la leyenda hay que llamar a `ax.legend`, se pasen o no las opciones de etiqueta al trazar los datos.

Marcas, etiquetas y leyendas

A través de métodos aplicados a los objetos de eje de matplotlib se puede acceder a la mayoría de los tipos de decoraciones de gráficos. Entre ellos se incluyen métodos como `xlim`, `xticks` y `xticklabels`, que controlan el rango del gráfico, las ubicaciones de las marcas y las etiquetas de las marcas, respectivamente. Se pueden utilizar de dos maneras:

- Si se les llama sin argumentos, devuelven el valor de parámetro actual (por ejemplo, `ax.xlim()` devuelve el rango de trazado actual del eje x).
- Si se les llama con parámetros, se configura el valor del parámetro (por ejemplo, `ax.xlim([0, 10])` configura el rango del eje x de 0 a 10).

Todos estos métodos actúan sobre el `AxesSubplot` que esté activo o haya sido creado más recientemente. Cada uno corresponde a dos métodos del mismo objeto de subgráfico; en el caso de `xlim`, estos métodos son `ax.get_xlim` y `ax.set_xlim`.

Configuración del título, las etiquetas de los ejes, las marcas y las etiquetas de marca

Para ilustrar la personalización de los ejes, creamos una sencilla figura con el trazado de un paseo aleatorio (figura 9.8):

```
In [40]: fig, ax = plt.subplots()
```

```
In [41]: ax.plot(np.random.standard_normal(1000).cumsum());
```

Para cambiar los marcadores del eje x, es más fácil utilizar `set_xticks` y `set_xticklabels`. El primero indica a matplotlib dónde colocar las marcas a lo largo del rango de datos; por defecto estas ubicaciones serán también las etiquetas. Pero podemos configurar cualesquiera otros valores como etiquetas utilizando `set_xticklabels`:

```
In [42]: ticks = ax.set_xticks([0, 250, 500, 750, 1000])
```

```
In [43]: labels = ax.set_xticklabels(["one", "two", "three",  
"four", "five"],
```

```
....:           rotation=30, fontsize=8)
```

La opción `rotation` configura las etiquetas de marca x con una rotación de 30 grados. Por último, `set_xlabel` asigna un nombre al eje x, y `set_title` es el título del subgráfico (véase el resultado en la figura 9.9):

```
In [44]: ax.set_xlabel("Stages")
```

```
Out[44]: Text(0.5, 6.666666666666652, 'Stages')
```

```
In [45]: ax.set_title("My first matplotlib plot")
```

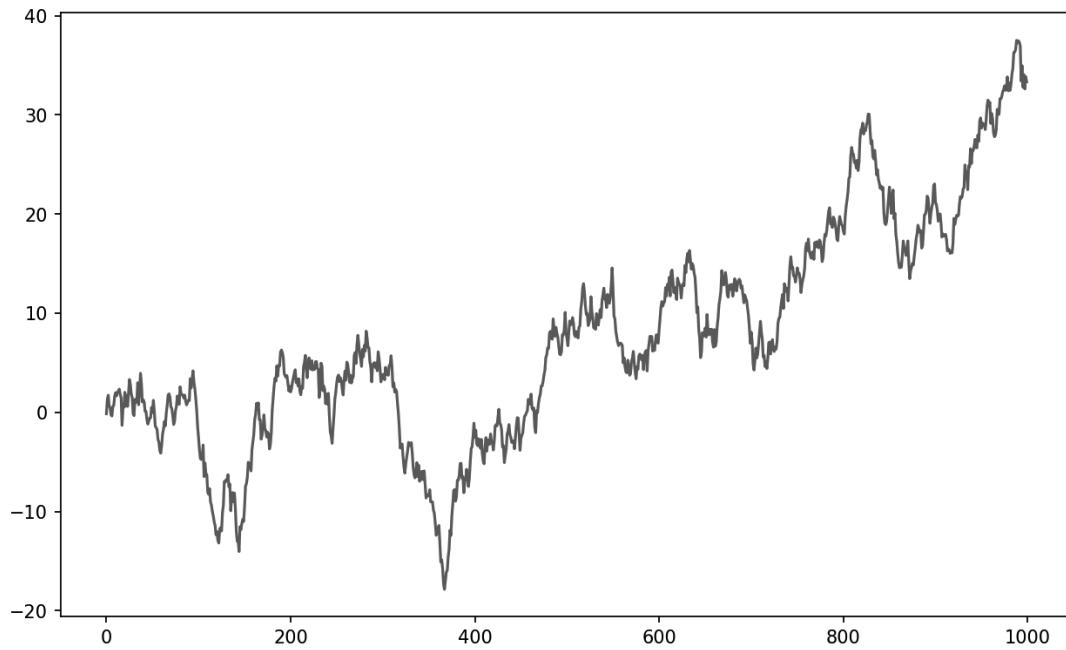


Figura 9.8. Sencillo gráfico para ilustrar los marcadores del eje x (con etiquetas predeterminadas).

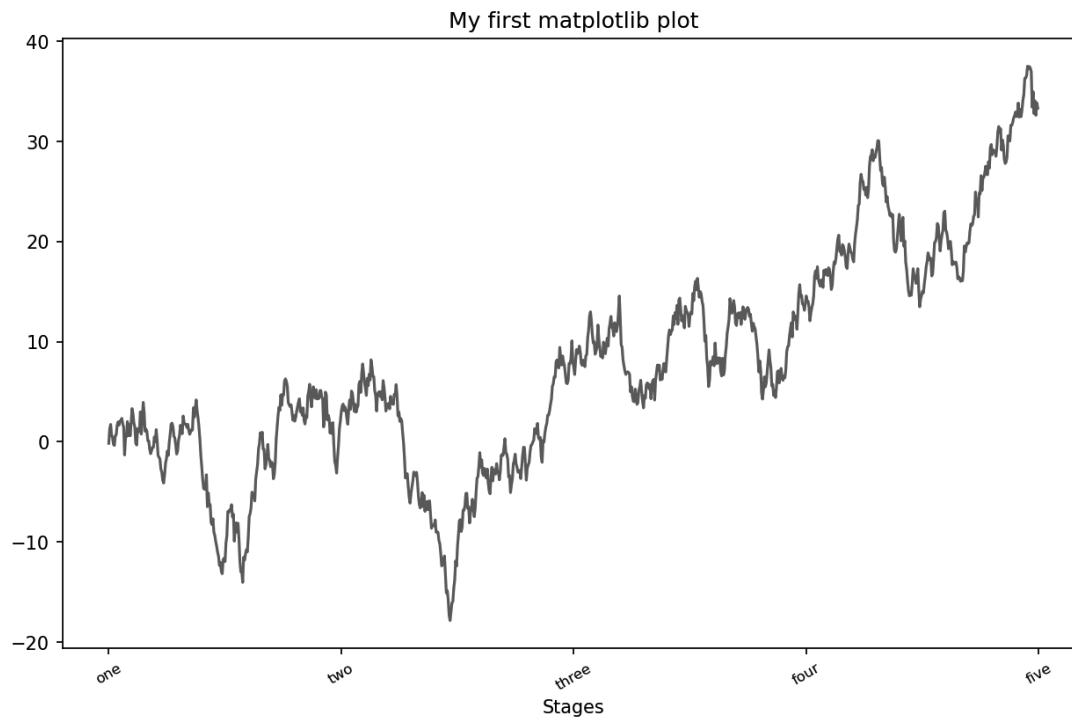


Figura 9.9. Sencillo gráfico para ilustrar la personalización de las marcas del eje x.

Para modificar el eje x se sigue el mismo proceso, sustituyendo y por x en este ejemplo. La clase de ejes tiene un método set que permite configurar en lote las propiedades del gráfico. Utilizando el ejemplo anterior, podríamos haber escrito:

```
ax.set(title="My first matplotlib plot", xlabel="Stages")
```

Incorporación de leyendas

Las leyendas son otro elemento fundamental para identificar elementos de gráfico. Hay un par de maneras de añadirlas, siendo la más sencilla pasar el argumento label al añadir cada parte del gráfico:

```
In [46]: fig, ax = plt.subplots()  
In [47]: ax.plot(np.random.randn(1000).cumsum(),  
color="black", label="one");  
In [48]: ax.plot(np.random.randn(1000).cumsum(),  
color="black", linestyle="dashed",  
....: label="two");  
In [49]: ax.plot(np.random.randn(1000).cumsum(),  
color="black", linestyle="dotted",  
....: label="three");
```

Una vez hecho esto, se puede llamar a `ax.legend()` para crear una leyenda automáticamente. El gráfico resultante aparece en la figura 9.10:

```
In [50]: ax.legend()
```

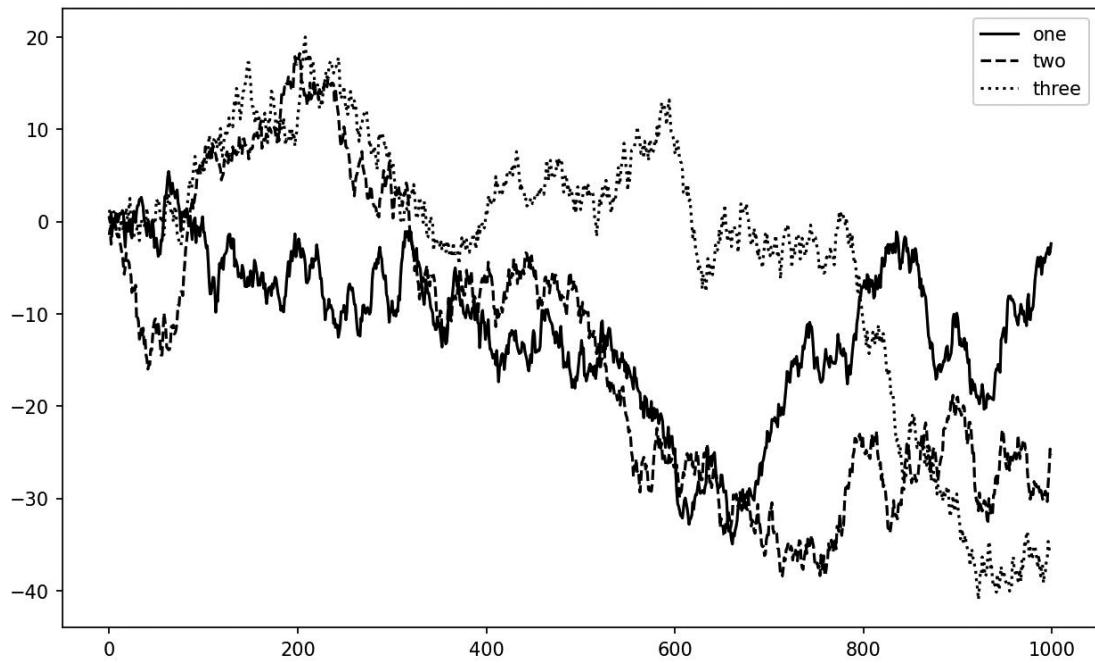


Figura 9.10. Sencillo gráfico con tres líneas y leyenda.

El método `legend` tiene otras opciones diversas para el argumento de ubicación `loc`. En el docstring se puede encontrar más información sobre esto (con `ax.legend?`).

La opción de leyenda `loc` le indica a `matplotlib` dónde colocar el gráfico. El valor predeterminado es “best”, que intenta elegir una ubicación lo más alejada posible. Para excluir uno o varios elementos de la leyenda, no pasamos ninguna etiqueta o bien `label="nolegend"`.

Anotaciones y dibujos en un subgráfico

Además de los tipos de gráfico estándares, es posible que surja la necesidad de dibujar anotaciones de gráfico propias, que podrían ser texto, flechas u otras formas. Esto puede hacerse utilizando las funciones `text`, `arrow` y `annotate`. `text` dibuja texto en las coordenadas indicadas (`x`, `y`) del gráfico con un estilo personalizado opcional:

```
ax.text(x, y, "Hello world!",
family="monospace", fontsize=10)
```

Las anotaciones pueden dibujar texto y flechas adecuadamente colocadas. Como ejemplo, vamos a trazar el precio de cierre del índice S&P 500 desde 2007 (obtenido en Yahoo! Finance) y a anotarlo con algunas de las fechas importantes desde la crisis financiera de 2008-2009. Se puede ejecutar este ejemplo de código en una sola celda en un Jupyter notebook. Véase el resultado en la figura 9.11:

```
from datetime import datetime
fig, ax = plt.subplots()
data = pd.read_csv("examples/spx.csv", index_col=0,
parse_dates=True)
spx = data["SPX"]

spx.plot(ax=ax, color="black")
crisis_data = [
(datetime(2007, 10, 11), "Peak of bull market"),
(datetime(2008, 3, 12), "Bear Stearns Fails"),
(datetime(2008, 9, 15), "Lehman Bankruptcy")]

]

for date, label in crisis_data:
ax.annotate(label, xy=(date, spx.asof(date) + 75),
xytext=(date, spx.asof(date) + 225),
arrowprops=dict(facecolor="black", headwidth=4, width=2,
headlength=4),
horizontalalignment="left", verticalalignment="top")

# Nos centramos en 2007-2010
ax.set_xlim(["1/1/2007", "1/1/2011"])
ax.set_ylim([600, 1800])

ax.set_title("Important dates in the 2008-2009 financial
crisis")
```

Hay un par de puntos importantes que destacar en este gráfico. El método `ax.annotate` puede dibujar etiquetas en las coordenadas indicadas x e y. Utilizamos los métodos `set_xlim` y `set_ylim` para configurar

manualmente los límites de inicio y fin para el gráfico, en lugar de utilizar el valor predeterminado de matplotlib. Finalmente, `ax.set_title` añade un título principal al gráfico.

En la galería de matplotlib en línea se pueden consultar muchos más ejemplos de anotación.

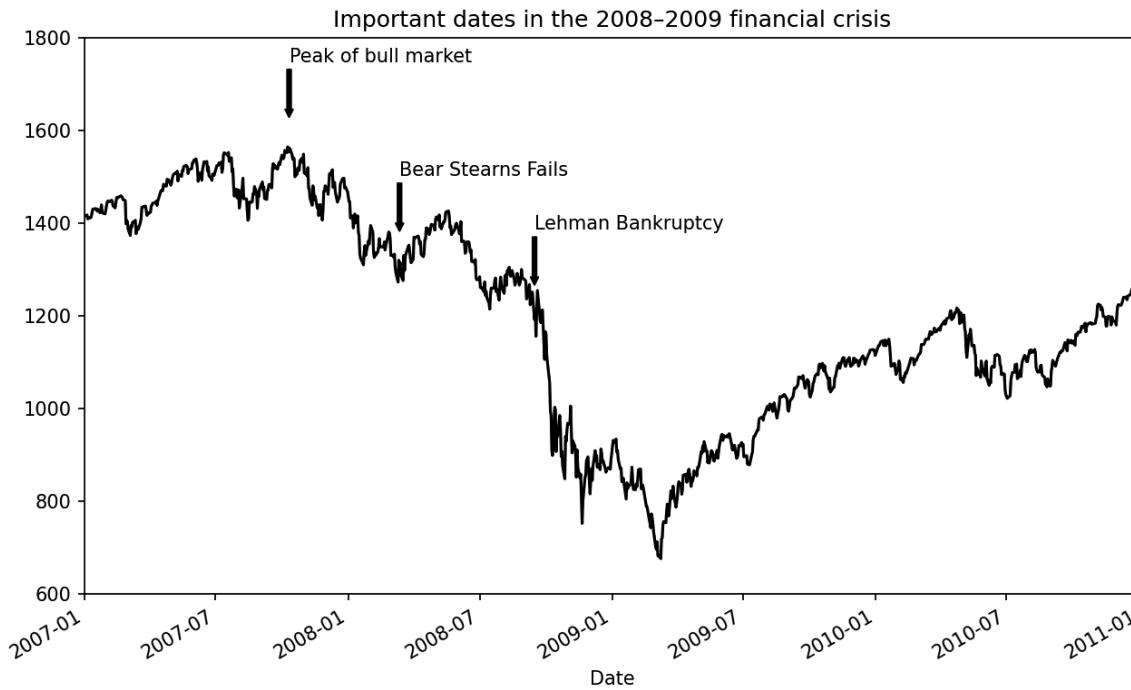


Figura 9.11. Fechas importantes de la crisis financiera de 2008-2009.

Dibujar formas requiere un poco de más atención. matplotlib tiene objetos que representan muchas de las formas habituales, las denominadas figuras geométricas. Algunas, como `Rectangle` y `Circle`, se encuentran en `matplotlib.pyplot`, pero el conjunto completo está en `matplotlib.patches`.

Para añadir una forma a un gráfico, se crea primero el objeto figura geométrica y después se añade al subgráfico `ax` pasándola a `ax.add_patch` (figura 9.12):

```
fig, ax = plt.subplots()
rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color="black",
alpha=0.3)
circ = plt.Circle((0.7, 0.2), 0.15, color="blue", alpha=0.3)
```

```

pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]],  

                   color="green", alpha=0.5)  
  

ax.add_patch(rect)
ax.add_patch(circ)
ax.add_patch(pgon)

```

Si echamos un vistazo a la implementación de muchos tipos de gráfico habituales, veremos que se montan a partir de figuras geométricas.

Almacenamiento de gráficos en archivo

Es posible guardar la figura activa en un archivo utilizando el método de instancia `savefig` del objeto figura. Por ejemplo, para guardar una versión SVG de una figura, solo hace falta escribir:

```
fig.savefig("figpath.svg")
```

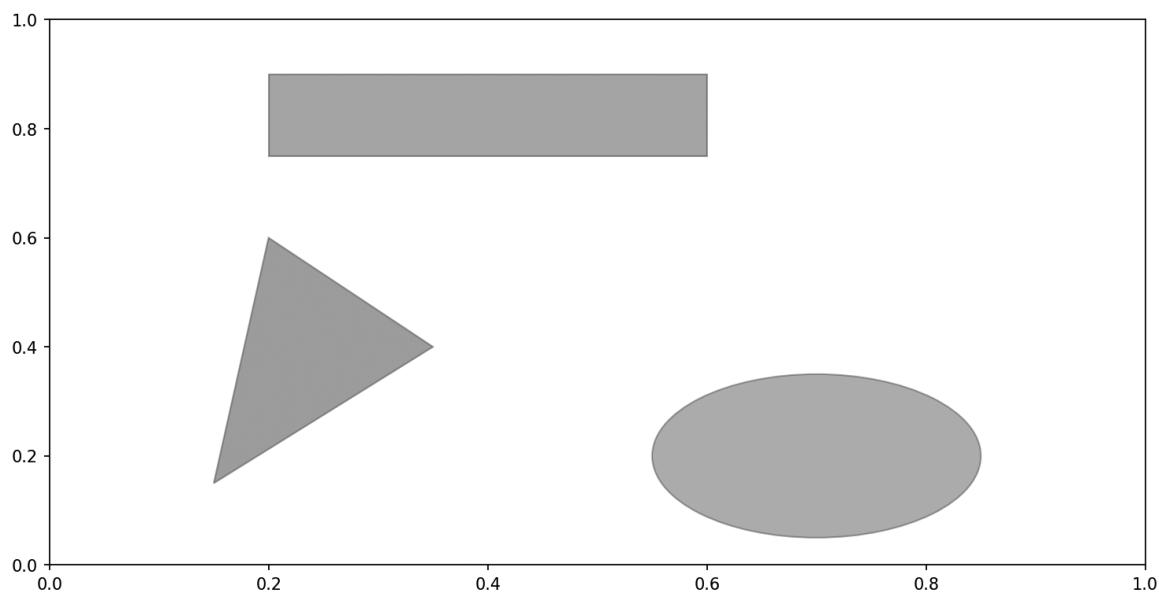


Figura 9.12. Visualización de datos compuesta por tres figuras geométricas distintas.

El tipo de archivo se deduce de la extensión del mismo. Así, si utilizamos .pdf, obtenemos un PDF. Una opción importante que suelo

utilizar para publicar gráficos es `dpi`, que controla la resolución de puntos por pulgada (`dpi`: *dots-per-inch*). Para obtener el mismo gráfico como PNG a 400 DPI, haríamos lo siguiente:

```
fig.savefig("figpath.png", dpi=400)
```

En la tabla 9.2 se puede consultar una lista con otras opciones adicionales para `savefig`. Si se desea la lista completa, basta con consultar el docstring de IPython o Jupyter.

Tabla 9.2. Algunas opciones de `fig.savefig`.

Argumento	Descripción
<code>fname</code>	Cadena de texto que contiene una ruta de archivo o un objeto de tipo archivo Python. El formato de figura se deduce de la extensión (por ejemplo, <code>.pdf</code> para PDF o <code>.png</code> para PNG).
<code>dpi</code>	La resolución de la figura en puntos por pulgada; su valor predeterminado es 100 en IPython o 72 en Jupyter, pero puede configurarse.
<code>facecolor</code> , <code>edgecolor</code>	El color del fondo de la figura fuera de los subgráficos; por defecto es "w" (white: blanco).
<code>format</code>	El formato de archivo explícito que se va a utilizar ("png", "pdf", "svg", "ps", "eps", etc.).

Configuración de matplotlib

El paquete matplotlib viene configurado por defecto con esquemas de color y valores orientados principalmente a la preparación de figuras para su publicación. Por suerte, casi todo el comportamiento predeterminado se puede personalizar mediante parámetros globales que controlan el tamaño de la figura, el espacio de subgráficos, los colores, los tamaños de fuente, los estilos de cuadrícula, etc. Un modo de modificar la configuración con código desde Python es utilizando el método `rc`; por ejemplo, para establecer que el tamaño de figura predeterminado global sea de 10 x 10, podríamos escribir:

```
plt.rc("figure", figsize=(10, 10))
```

Todos los valores de configuración actuales están en el diccionario `plt.rcParams`, y se pueden restaurar a sus valores por defecto llamando a la función `plt.rcdefaults()`.

El primer argumento para `rc` es el componente que conviene personalizar, como “`figure`”, “`axes`”, “`xtick`”, “`ytick`”, “`grid`”, “`legend`” o muchos otros. A continuación puede venir una secuencia de argumentos de palabra clave indicando los parámetros nuevos. Una forma cómoda de anotar las opciones del programa es como un diccionario:

```
plt.rc("font", family="monospace", weight="bold", size=8)
```

Si se desea una mayor personalización y se quiere disponer de una lista con todas las opciones, matplotlib incluye un archivo de configuración `matplotlibrc` en el directorio `matplotlib/mpl-data`. Si personalizamos este archivo y lo colocamos en el directorio de inicio denominado `.matplotlibrc`, se cargará cada vez que se utilice matplotlib.

Como veremos en la próxima sección, el paquete seaborn tiene varios temas o estilos internos que emplean internamente el sistema de configuración de matplotlib.

9.2 Realización de gráficos con pandas y seaborn

En esencia, matplotlib puede ser una herramienta de nivel bastante bajo. Los gráficos se crean a partir de sus componentes básicos: la visualización de datos (es decir, el tipo de gráfico: líneas, barras, cajas, dispersión, contorno, etc.), la leyenda, el título, las etiquetas de marcas y otras anotaciones.

En pandas, podríamos tener varias columnas de datos, además de etiquetas de fila y columna. El mismo pandas posee métodos internos que simplifican la creación de visualizaciones a partir de objetos `DataFrame` y `Series`. Otra librería es `seaborn` (<https://seaborn.pydata.org>), una

librería de gráficos estadísticos de alto nivel integrada en matplotlib. seaborn simplifica la creación de muchos tipos de visualización habituales.

Gráficos de líneas

Los objetos Series y DataFrame tienen un atributo `plot` para crear algunos tipos de gráfico básicos. De forma predeterminada, `plot()` crea gráficos de líneas (ver la figura 9.13):

```
In [61]: s = pd.Series(np.random.standard_normal(10).cumsum(),
index=np.arange(0, 100, 10))

In [62]: s.plot()
```

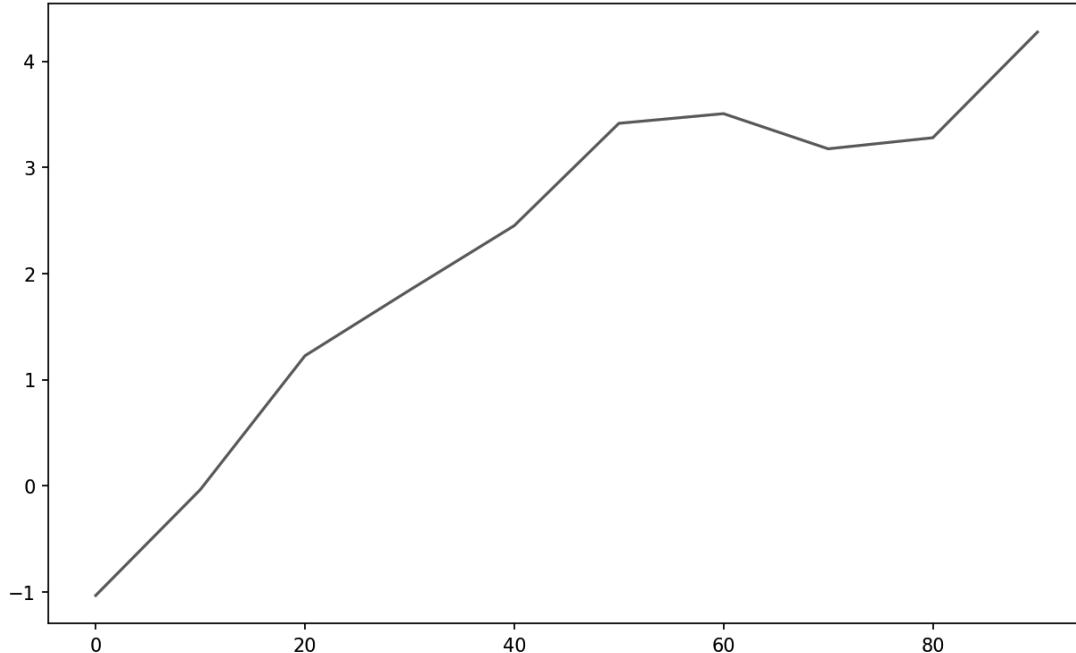


Figura 9.13. Sencillo gráfico Series.

El índice del objeto Series se pasa a matplotlib para trazar en el eje x, aunque esto se puede deshabilitar pasando `use_index=False`. Las marcas y los límites del eje x se pueden ajustar con las opciones `xticks` y `xlim`, y los del eje y con `yticks` y `ylim`, respectivamente. La tabla 9.3 incluye un

listado parcial de opciones de trazado. A lo largo de esta sección comentaré algunos más, y dejaré el resto para su exploración por parte de los lectores.

Tabla 9.3. Argumentos de método Series.plot.

Argumento	Descripción
label	Etiqueta para la leyenda del gráfico.
ax	Objeto de subgráfico de matplotlib según el cual trazar; si no se pasa nada, utiliza el subgráfico de matplotlib activo.
style	Estilo de la cadena de texto, como "ko-", que se le va a pasar a matplotlib.
alpha	La opacidad de relleno del gráfico (de 0 a 1).
kind	Puede ser "area", "bar", "barh", "density", "hist", "kde", "line" o "pie"; por defecto es "line".
figsize	Tamaño del objeto de figura que se va a crear.
logx	Pasa True para la escala logarítmica en el eje x; pasa "sym" para un logaritmo simétrico que permite valores negativos.
logy	Pasa True para la escala logarítmica en el eje y; pasa "sym" para un logaritmo simétrico que permite valores negativos.
title	Título que se va a utilizar para el gráfico.
use_index	Usa el índice del objeto para las etiquetas de marcas.
rot	Rotación de las etiquetas de marcas (0 a 360).
xticks	Valores a utilizar para las marcas en el eje x.
yticks	Valores a utilizar para las marcas en el eje y.
xlim	Límites del eje x (por ejemplo, [0, 10]).
ylim	Límites del eje y.
grid	Muestra la cuadrícula del eje (desactivado por defecto).

La mayor parte de los métodos de trazado de pandas aceptan un parámetro opcional `ax`, que puede ser un objeto subgráfico de matplotlib.

Esto proporciona una ubicación más flexible de los subgráficos en una disposición de cuadrícula.

El método `plot` de `DataFrame` traza cada una de sus columnas como una línea diferente en el mismo subgráfico, creando una leyenda de manera automática (véase la figura 9.14):

```
In [63]: df = pd.DataFrame(np.random.standard_normal((10, 4)).cumsum(0),
```

```
.....:         columns=[“A”, “B”, “C”, “D”],  
.....:         index=np.arange(0, 100, 10))
```

```
In [64]: plt.style.use('grayscale')
```

```
In [65]: df.plot()
```

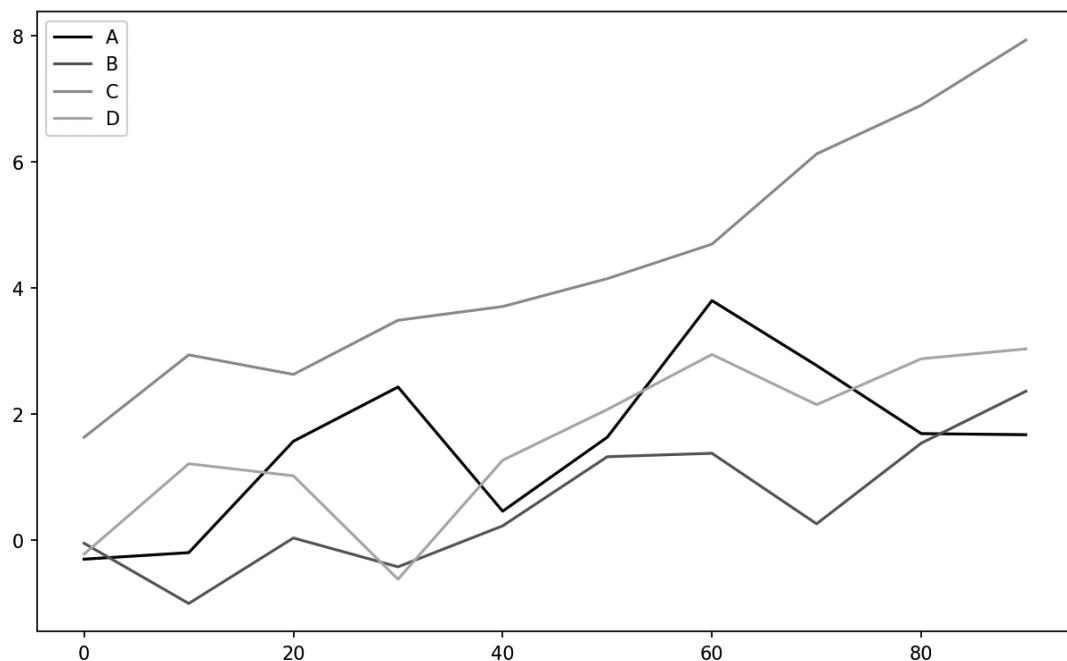


Figura 9.14. Sencillo gráfico `DataFrame`.



Aquí he utilizado `plt.style.use('grayscale')` para cambiar a un esquema de color más adecuado para publicaciones en blanco y negro, ya que algunos lectores no serán capaces de ver gráficos a todo color.

El atributo `plot` contiene una “familia” de métodos para distintos tipos de gráficos. Por ejemplo, `df.plot()` es equivalente a `df.plot.line()`. Exploraremos a continuación algunos de estos métodos.



Los argumentos de palabra clave adicionales a `plot` se pasan a la función de trazado de `matplotlib` respectiva, de forma que se pueden personalizar estos gráficos aprendiendo más sobre la API de `matplotlib`.

`DataFrame` tiene distintas opciones que permiten una cierta flexibilidad en la forma de manejar las columnas, por ejemplo, si trazarlas todas en el mismo subgráfico o crear subrgráficos distintos. La tabla 9.4 ofrece más argumentos de este objeto.

Tabla 9.4. Argumentos de trazado específicos de DataFrame.

Argumento	Descripción
<code>subplots</code>	Traza cada columna del objeto <code>DataFrame</code> en un subgráfico distinto.
<code>layouts</code>	Tuplas de 2 (filas, columnas) que ofrecen disposición de subgráficos.
<code>sharex</code>	Si <code>subplots=True</code> , comparte el mismo eje x, enlazando marcas y límites.
<code>sharey</code>	Si <code>subplots=True</code> , comparte el mismo eje y.
<code>legend</code>	Añade una leyenda en el subgráfico (<code>True</code> por defecto).
<code>sort_columns</code>	Traza las columnas en orden alfabético; usa por defecto el orden de columnas existente.



Si desea saber más sobre los gráficos de series temporales, consulte el capítulo 11.

Gráficos de barras

`plot.bar()` y `plot.bart()` crean gráficos de barras verticales y horizontales, respectivamente. En este caso, el índice del objeto `Series` o

DataFrame se utilizará como las marcas x (bar) o y (barh) (véase la figura 9.15):

```
In [66]: fig, axes = plt.subplots(2, 1)
```

```
In [67]: data = pd.Series(np.random.uniform(size=16),  
index=list("abcdefghijklmno"))
```

```
In [68]: data.plot.bar(ax=axes[0], color="black", alpha=0.7)  
Out[68]: <AxesSubplot:>
```

```
In [69]: data.plot.barh(ax=axes[1], color="black",  
alpha=0.7)
```

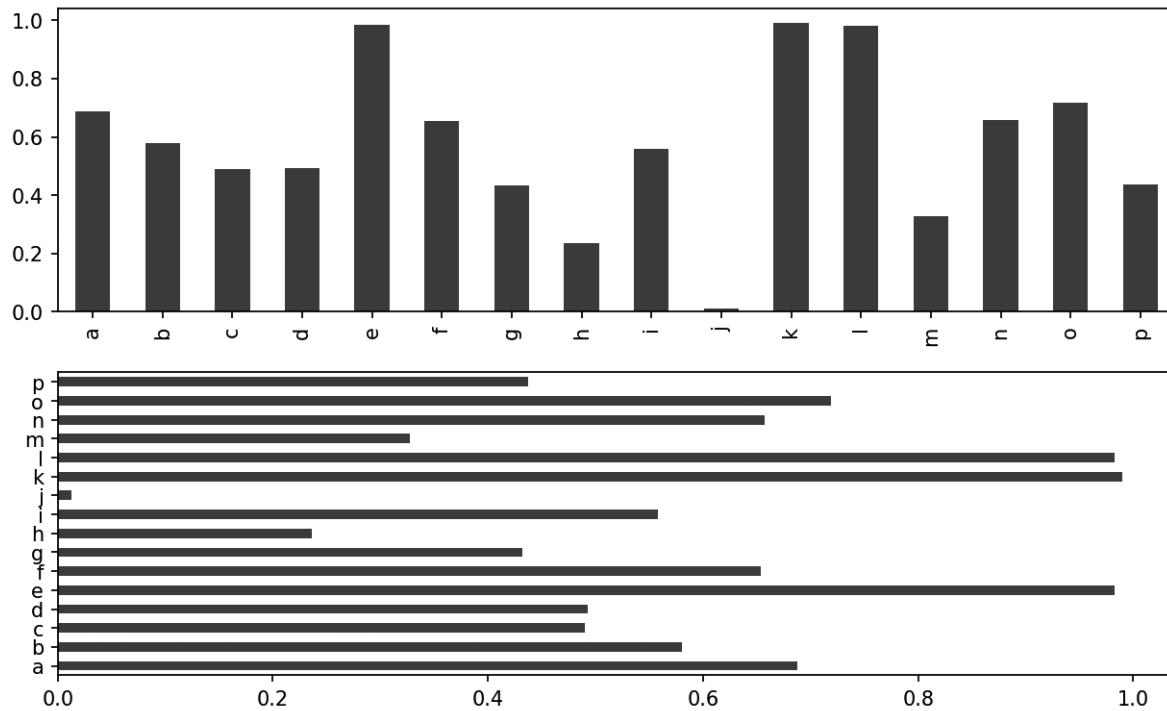


Figura 9.15. Gráfico de barras horizontales y verticales.

Con un dataframe, los gráficos de barras agrupan los valores de cada fila en barras, una al lado de la otra, para cada valor. Véase la figura 9.16:

```
In [71]: df = pd.DataFrame(np.random.uniform(size=(6, 4)),  
....: index=["one", "two", "three", "four", "five",  
....: "six"],  
....: columns=pd.Index(["A", "B", "C", "D"],
```

```
name="Genus"))
```

```
In [72]: df
```

```
Out[72]:
```

Genus	A	B	C	D
one	0.370670	0.602792	0.229159	0.486744
two	0.420082	0.571653	0.049024	0.880592
three	0.814568	0.277160	0.880316	0.431326
four	0.374020	0.899420	0.460304	0.100843
five	0.433270	0.125107	0.494675	0.961825
six	0.601648	0.478576	0.205690	0.560547

```
In [73]: df.plot.bar()
```

Podemos observar que el nombre “Genus” de las columnas del dataframe se utiliza para dar título a la leyenda.

Creamos gráficos de barras apiladas a partir de un dataframe pasando stacked=True, lo que da como resultado que el valor de cada fila sea apilado en horizontal (ver figura 9.17):

```
In [75]: df.plot.bah(stacked=True, alpha=0.5)
```

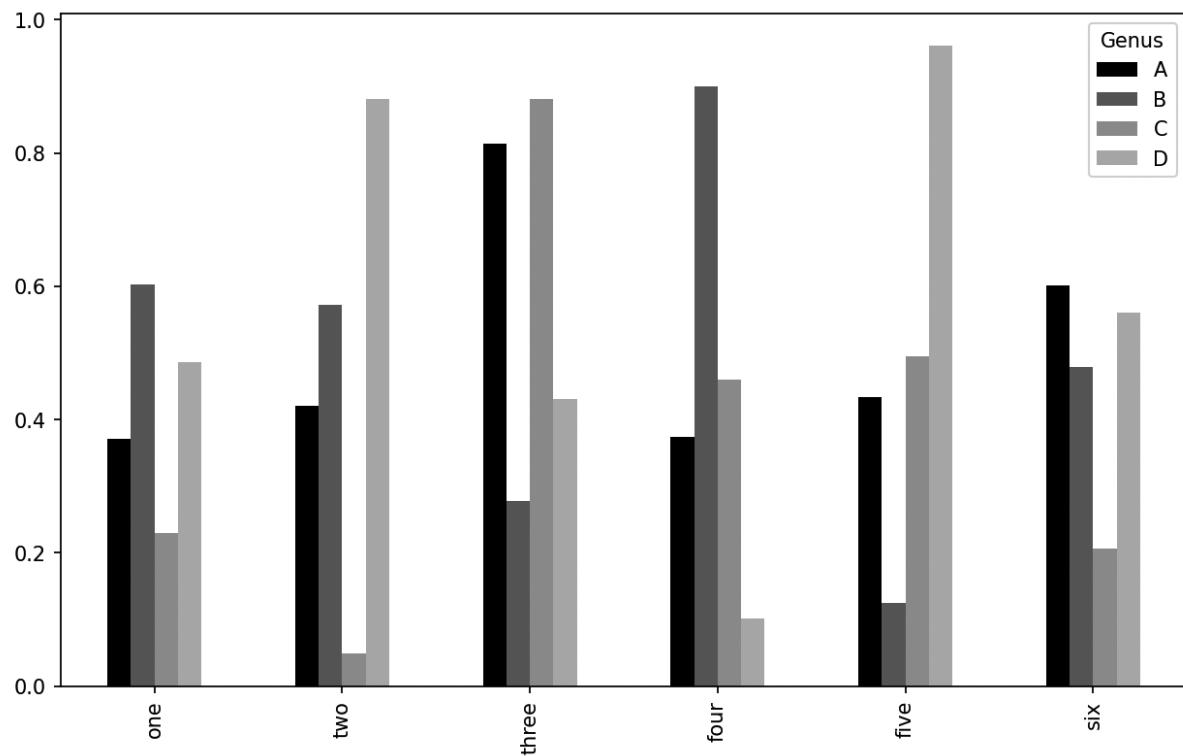


Figura 9.16. Gráfico de barras de un dataframe.

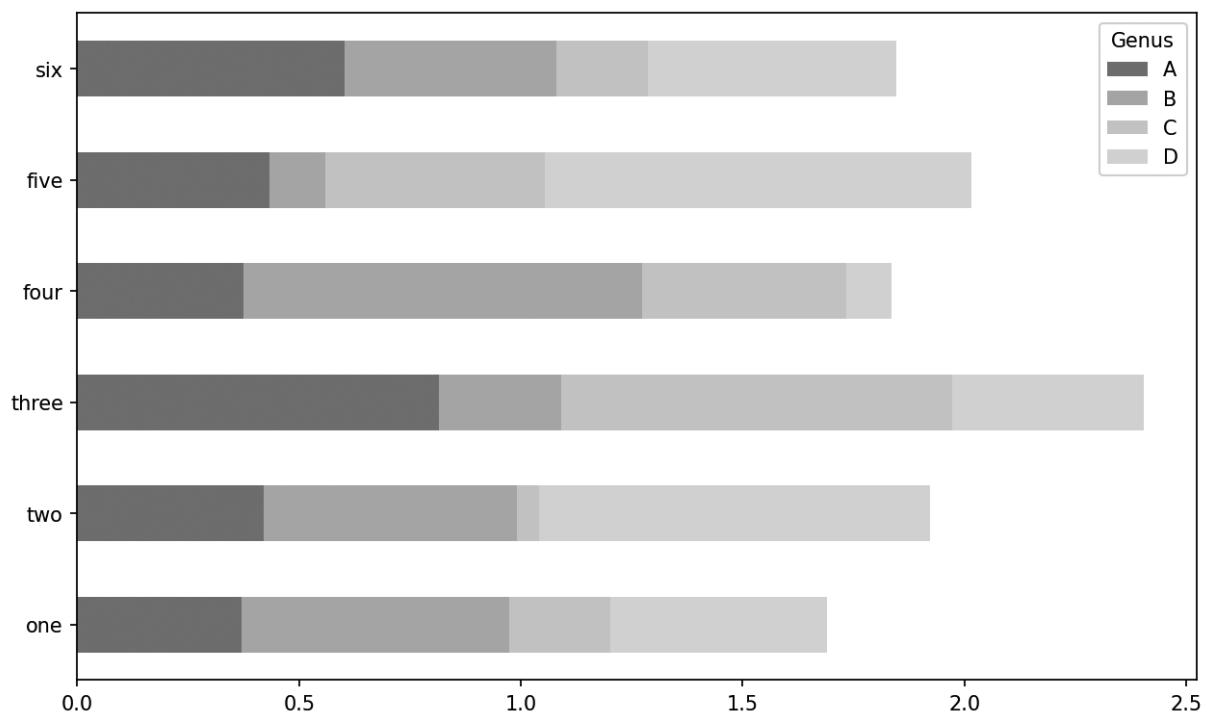


Figura 9.17. Gráfico de barras apiladas de un dataframe.



Una receta práctica para los gráficos de barras es visualizar la frecuencia de valor de una serie utilizando `value_counts(): s.value_counts().plot.bar()`.

Veamos un conjunto de datos de ejemplo sobre propinas en restaurantes. Supongamos que queremos crear un gráfico de barras apiladas que muestre el porcentaje de puntos de datos para cada tamaño de grupo y para cada día. Cargo los datos utilizando `read_csv` y hago una tabulación cruzada por día y tamaño de grupo. La función `pandas.crosstab` es una forma cómoda de calcular una sencilla tabla de frecuencias a partir de dos columnas de un `dataframe`:

```
In [77]: tips = pd.read_csv("examples/tips.csv")
```

```
In [78]: tips.head()
```

```
Out[78]:
```

	total_bill	tip	smoker	day	time	size
0	16.99	1.01		Sun	Dinner	2
1	10.34	1.66		Sun	Dinner	3
2	21.01	3.50		Sun	Dinner	3
3	23.68	3.31		Sun	Dinner	2
4	24.59	3.61		Sun	Dinner	4

```
In [79]: party_counts = pd.crosstab(tips["day"], tips["size"])
```

```
In [80]: party_counts = party_counts.reindex(index=["Thur", "Fri", "Sat", "Sun"])
```

```
In [81]: party_counts
```

```
Out[81]:
```

	1	2	3	4	5	6
size						
day						
Thur	1	48	4	5	1	3
Fri	1	16	1	1	0	0

Sat	2	53	18	13	1	0
Sun	0	39	15	18	3	1

Como no hay muchos grupos de una y seis personas, aquí los quito:

```
In [82]: party_counts = party_counts.loc[:, 2:5]
```

Después normalizo para que cada fila sume 1 y creo el gráfico (ver figura 9.18):

```
# Normaliza para sumar 1
```

```
In [83]: party_pcts =  
party_counts.div(party_counts.sum(axis="columns"),  
.....: axis="index")
```

```
In [84]: party_pcts  
Out[84]:
```

	2	3	4	5
size				
day				
Thur	0.827586	0.068966	0.086207	0.017241
Fri	0.888889	0.055556	0.055556	0.000000
Sat	0.623529	0.211765	0.152941	0.011765
Sun	0.520000	0.200000	0.240000	0.040000

```
In [85]: party_pcts.plot.bar(stacked=True)
```

Puede comprobarse que los tamaños de grupo parecen aumentar el fin de semana en este conjunto de datos.

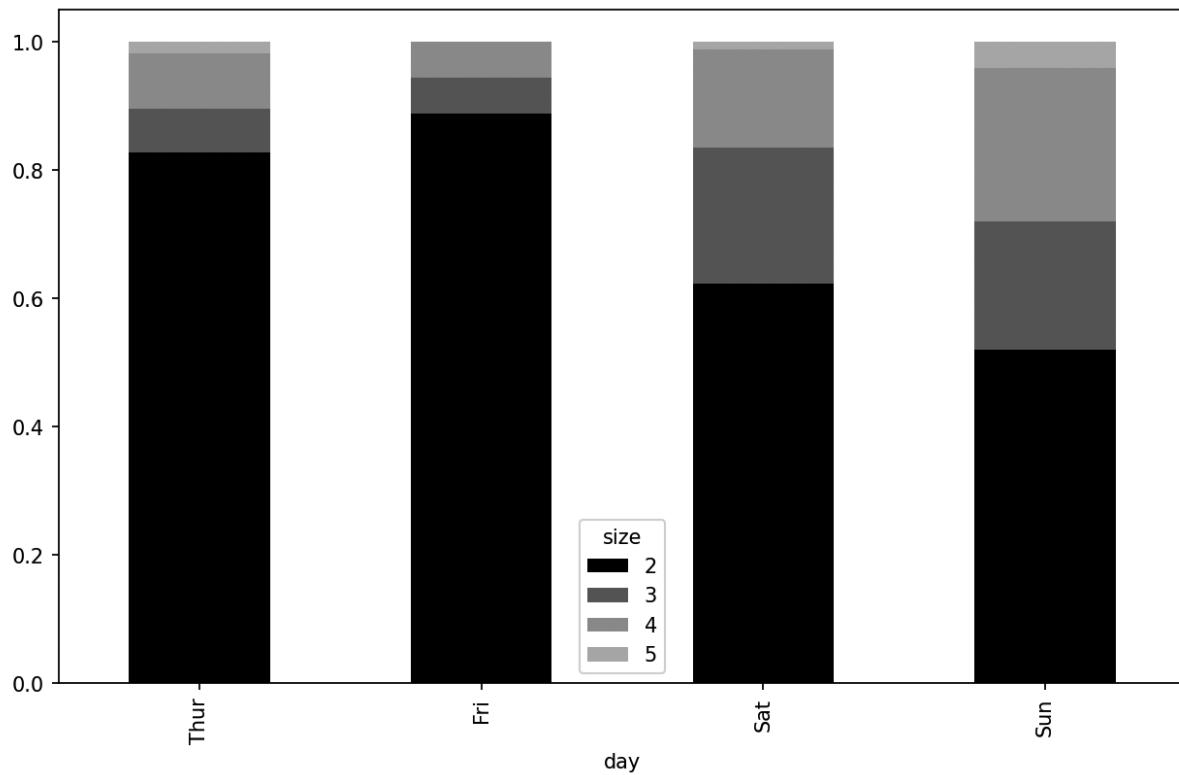


Figura 9.18. Fracción de grupos por tamaño dentro de cada día.

Con datos que requieren agregación o resumen antes de crear un gráfico, utilizar el paquete seaborn puede facilitar las cosas mucho (se instala con `conda install seaborn`). Veamos ahora el porcentaje de propinas diario con seaborn (véase el gráfico resultante en la figura 9.19):

```
In [87]: import seaborn as sns
```

```
In [88]: tips["tip_pct"] = tips["tip"] / (tips["total_bill"] - tips["tip"])
```

```
In [89]: tips.head()
```

```
Out[89]:
```

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01		Sun	Dinner	2	0.063204
1	10.34	1.66		Sun	Dinner	3	0.191244
2	21.01	3.50		Sun	Dinner	3	0.199886
3	23.68	3.31		Sun	Dinner	2	0.162494
4	24.59	3.61		Sun	Dinner	4	0.172069

```
In [90]: sns.barplot(x="tip_pct", y="day", data=tips, orient="h")
```

Las funciones de gráficos en seaborn requieren un argumento `data`, que puede ser un dataframe pandas. Los demás argumentos se refieren a nombres de columnas. Como hay muchas observaciones de cada valor en el día, las barras son el valor medio de `tip_pct`. Las líneas negras dibujadas en las barras representan el intervalo de confianza del 95% (esto se puede configurar mediante argumentos opcionales).

`seaborn.barplot` tiene una opción `hue` que permite dividir por un valor categórico adicional (véase la figura 9.20):

```
In [92]: sns.barplot(x="tip_pct", y="day", hue="time", data=tips, orient="h")
```

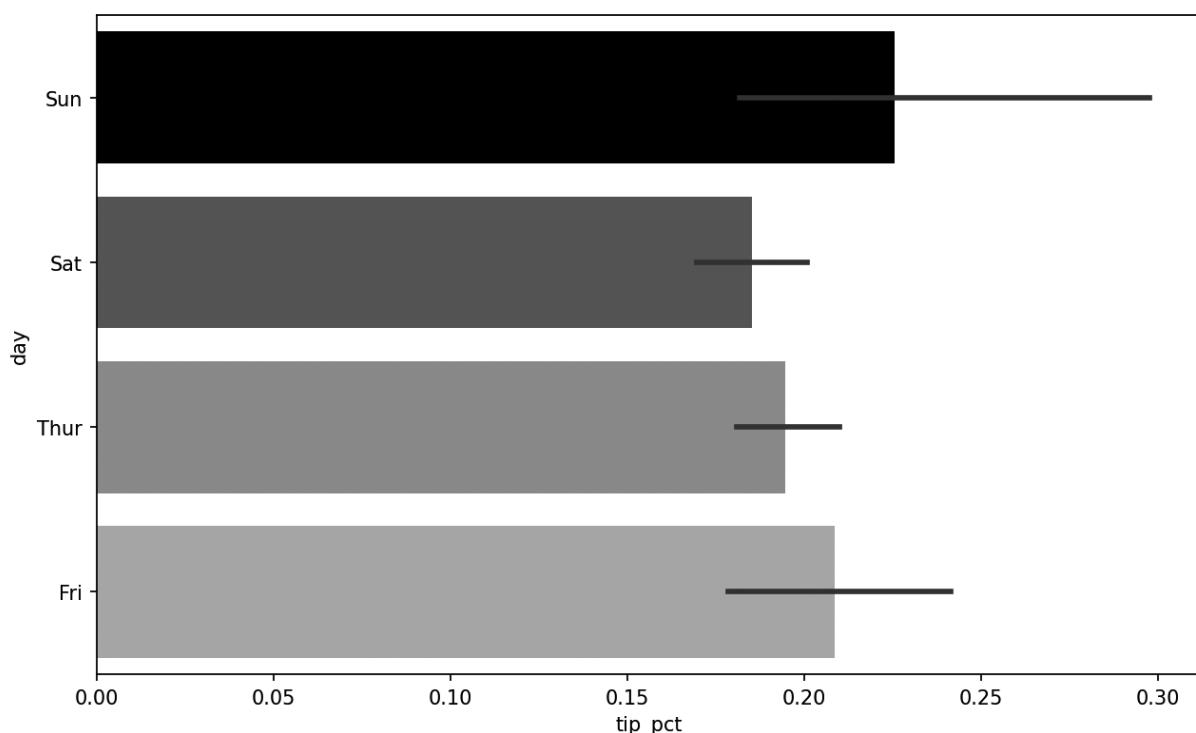


Figura 9.19. Porcentaje de propinas diario con barras de error.

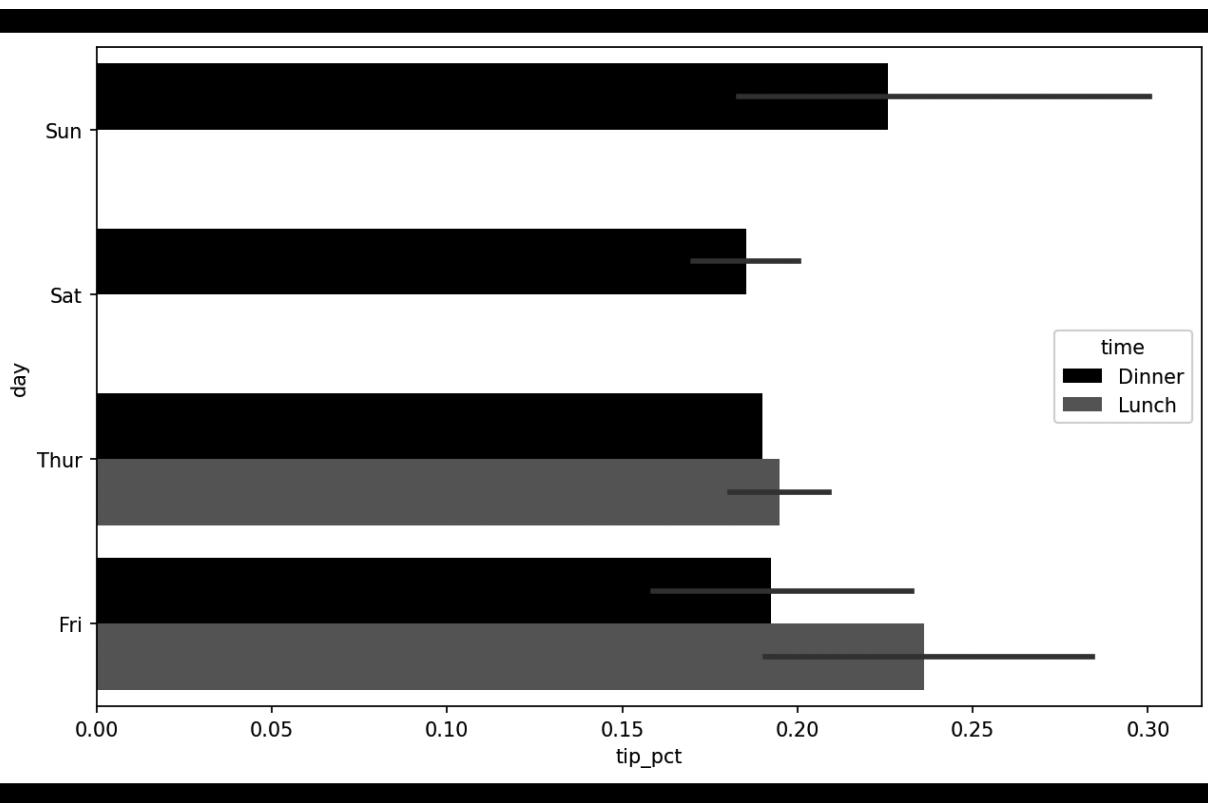


Figura 9.20. Porcentaje de propinas por día y hora.

Observamos que seaborn ha cambiado automáticamente la estética de los gráficos: la paleta de color predeterminada, el fondo del gráfico y los colores de las líneas de la cuadrícula. Se puede alternar entre las distintas apariencias de gráfico utilizando `seaborn.set_style`:

```
In [94]: sns.set_style("whitegrid")
```

Cuando se producen gráficos para su impresión en blanco y negro, puede resultar útil configurar una paleta de color de escalas de gris, del siguiente modo:

```
sns.set_palette("Greys_r")
```

Histogramas y gráficos de densidad

Un histograma es un tipo de gráfico de barras que ofrece una visualización discreta de la frecuencia de los valores. Los puntos de datos se

dividen en contenedores discretos y uniformemente espaciados, y se traza el número de puntos de datos de cada contenedor. Utilizando los datos de propinas anteriores, podemos crear un histograma de porcentajes de propina de la factura total empleando el método `plot.hist` de la serie (véase la figura 9.21):

```
In [96]: tips["tip_pct"].plot.hist(bins=50)
```

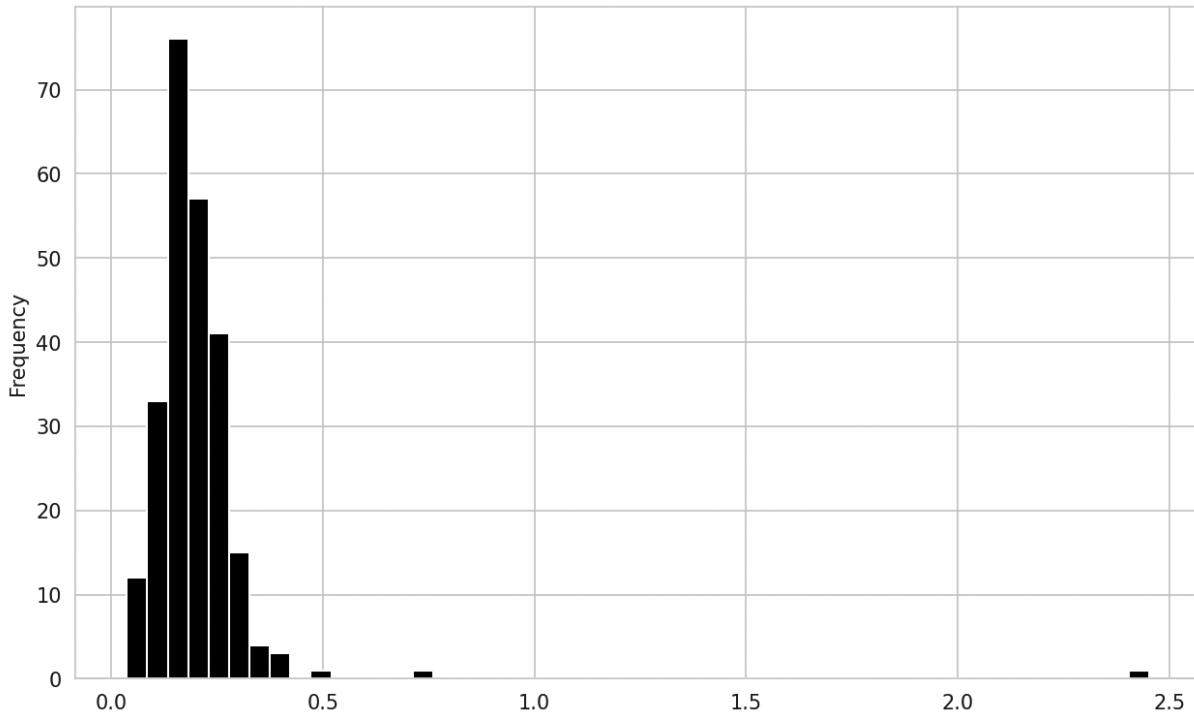


Figura 9.21. Histograma de porcentajes de propinas.

Un tipo de gráfico relacionado es el gráfico de densidad, formado por el cálculo de una estimación de una distribución continua de probabilidades que podría haber generado los datos observados. El procedimiento habitual es aproximar esta distribución como una mezcla de «kernels» (es decir, distribuciones más sencillas, como la normal). Así, los gráficos de densidad se conocen también como gráficos KDE (*Kernel Density Estimate*: estimación de densidad de kernel). Utilizar `plot.density` crea un gráfico de densidad empleando la estimación convencional de mezcla de distribuciones normales (figura 9.22):

```
In [98]: tips["tip_pct"].plot.density()
```

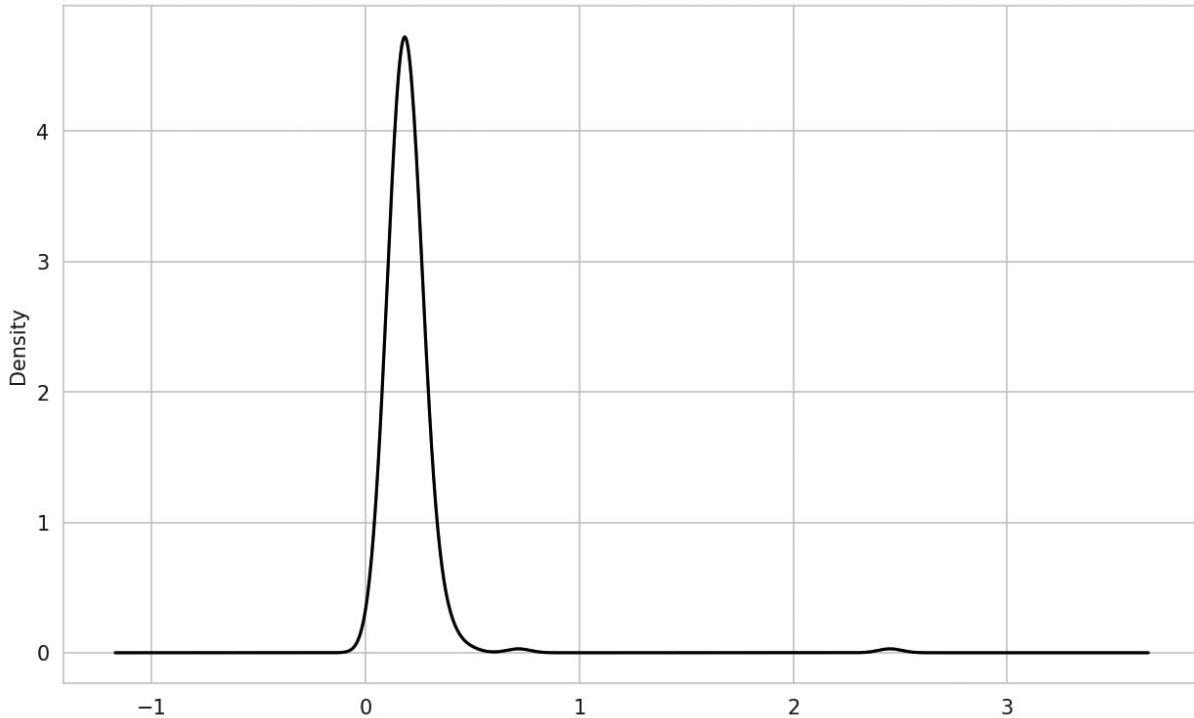


Figura 9.22. Gráfico de densidad de porcentajes de propinas.

Este tipo de gráficos requiere SciPy, de forma que si no está ya instalado, se puede hacer una pausa e instalarlo a continuación:

```
conda install scipy
```

La herramienta seaborn permite crear histogramas y gráficos de densidad de una forma aún más sencilla mediante su método `histplot`, capaz de trazar simultáneamente un histograma y una estimación de densidad continua. Como ejemplo, veamos una distribución bimodal formada por dibujos de dos distribuciones normales estándares distintas (figura 9.23):

```
In [100]: comp1 = np.random.standard_normal(200)
```

```
In [101]: comp2 = 10 + 2 * np.random.standard_normal(200)
```

```
In [102]: values = pd.Series(np.concatenate([comp1, comp2]))
```

```
In [103]: sns.histplot(values, bins=100, color="black")
```

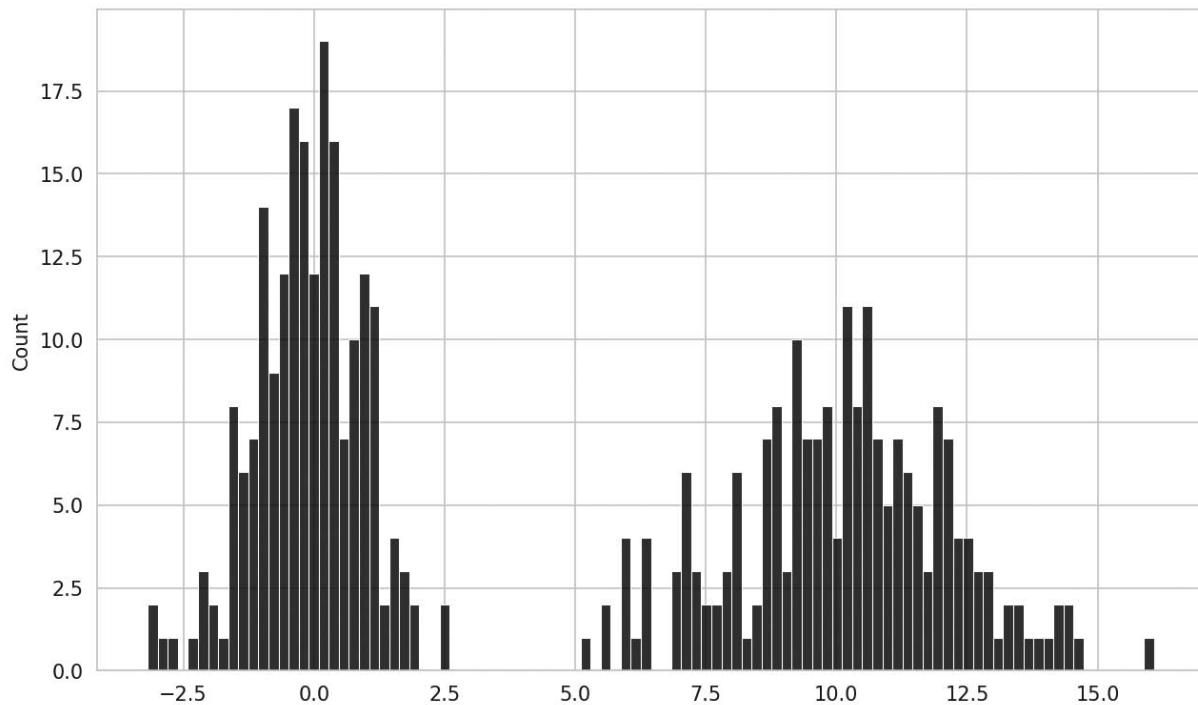


Figura 9.23. Histograma normalizado de mezcla normal.

Gráficos de dispersión o de puntos

Los gráficos de puntos o de dispersión pueden ser una forma útil de examinar las relaciones entre dos series de datos unidimensionales. Por ejemplo, aquí cargamos el conjunto de datos macrodata desde el proyecto statsmodels, seleccionamos algunas variables y después calculamos las diferencias logarítmicas:

```
In [104]: macro = pd.read_csv("examples/macrodata.csv")
In [105]: data = macro[["cpi", "m1", "tbilrate", "unemp"]]
In [106]: trans_data = np.log(data).diff().dropna()
In [107]: trans_data.tail()
Out[107]:
```

	cpi	m1	tbilrate	unemp
198	-0.007904	0.045361	-0.396881	0.105361
199	-0.021979	0.066753	-2.277267	0.139762

200	0.002340	0.010286	0.606136	0.160343
201	0.008419	0.037461	-0.200671	0.127339
202	0.008894	0.012202	-0.405465	0.042560

Después podemos utilizar el método `regplot` de `seaborn`, que crea un gráfico de dispersión y ajusta una línea de regresión lineal (figura 9.24):

```
In [109]: ax = sns.regplot(x="m1", y="unemp",
data=trans_data)
In [110]: ax.title("Changes in log(m1) versus log(unemp)")
```

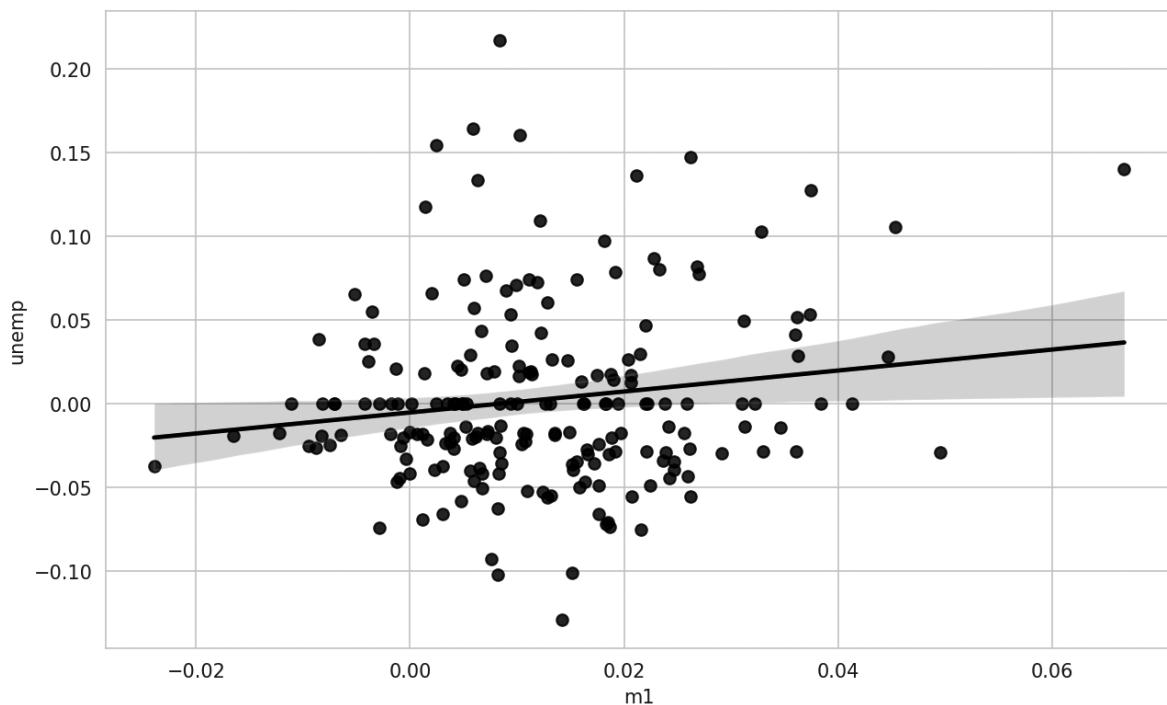


Figura 9.24. Un gráfico de dispersión/regresión seaborn.

En análisis de datos exploratorios, resulta útil poder mirar todos los gráficos de dispersión de entre un grupo de variables; esto se conoce como diagrama de pares o matriz de gráficos de dispersión. Crear un gráfico como este desde cero es costoso, por lo que `seaborn` tiene una conveniente función `pairplot`, que soporta la ubicación de histogramas o estimaciones de densidad de cada variable a lo largo de la diagonal (véase el gráfico resultante en la figura 9.25):

```
In [111]: sns.pairplot(trans_data, diag_kind="kde",
plot_kws={"alpha": 0.2})
```

Quizá algún lector haya reparado en el argumento `plot_kws`. Nos permite pasar opciones de configuración a las llamadas individuales a gráficos en los elementos fuera de la diagonal. En el docstring `seaborn.pairplot` se pueden encontrar detalladas más opciones de configuración.

Cuadrícula de facetas y datos categóricos

¿Qué pasa con los conjuntos de datos cuando tenemos dimensiones de agrupamiento adicionales? Una forma de visualizar datos con muchas variables categóricas es utilizar una cuadrícula de facetas. Se trata de una disposición bidimensional de gráficos en la que los datos se dividen a lo largo de los gráficos en cada eje basándose en los distintos valores de una determinada variable. `seaborn` tiene una útil función interna `cat` que simplifica la creación de muchos tipos de gráficos facetados divididos por variables categóricas (véase el gráfico resultante en la figura 9.26):

```
In [112]: sns.catplot(x="day", y="tip_pct", hue="time",
col="smoker",
.....: kind="bar", data=tips[tips.tip_pct < 1])
```

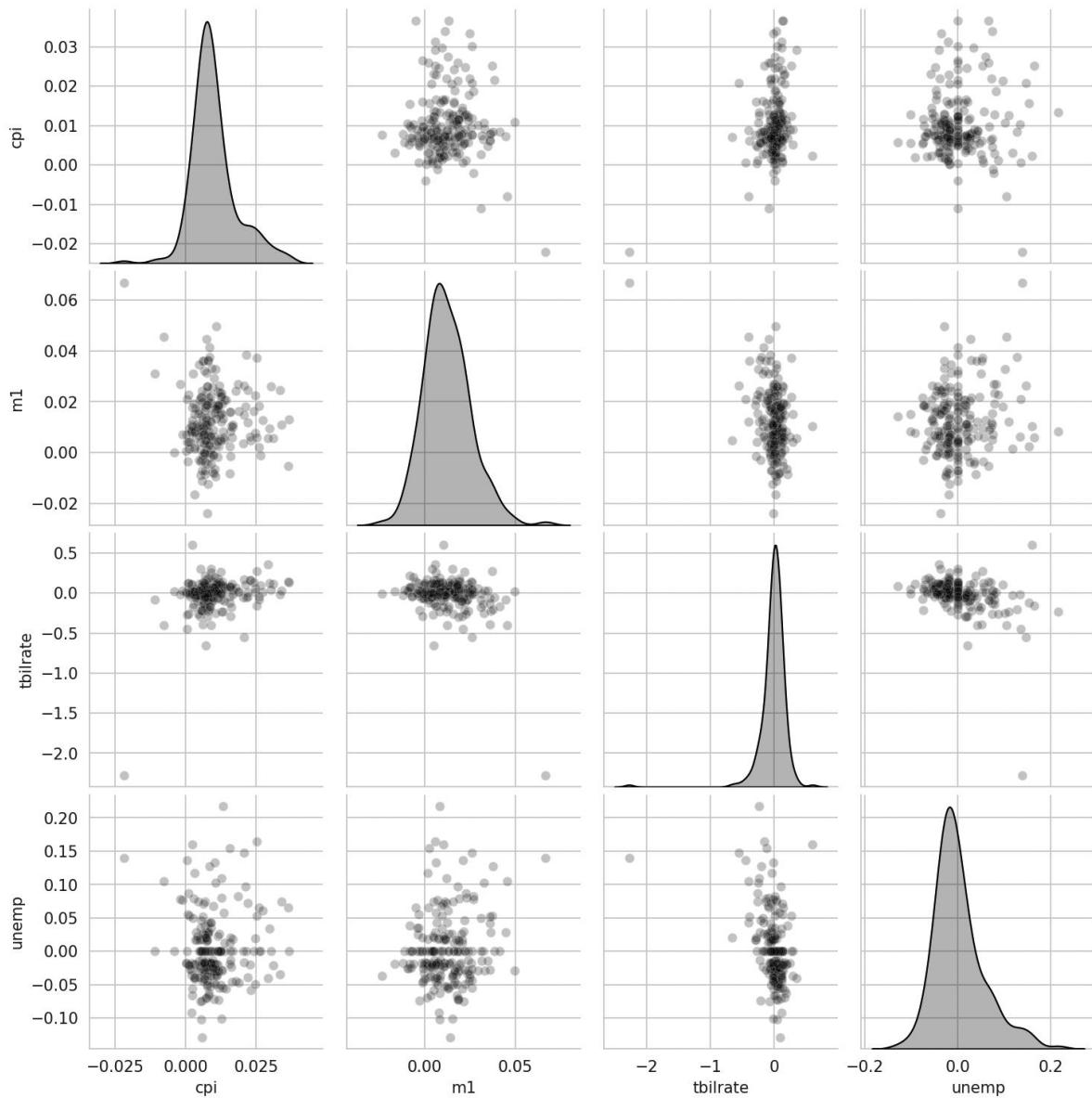


Figura 9.25. Matriz de gráficos de pares de datos macro de statsmodels.

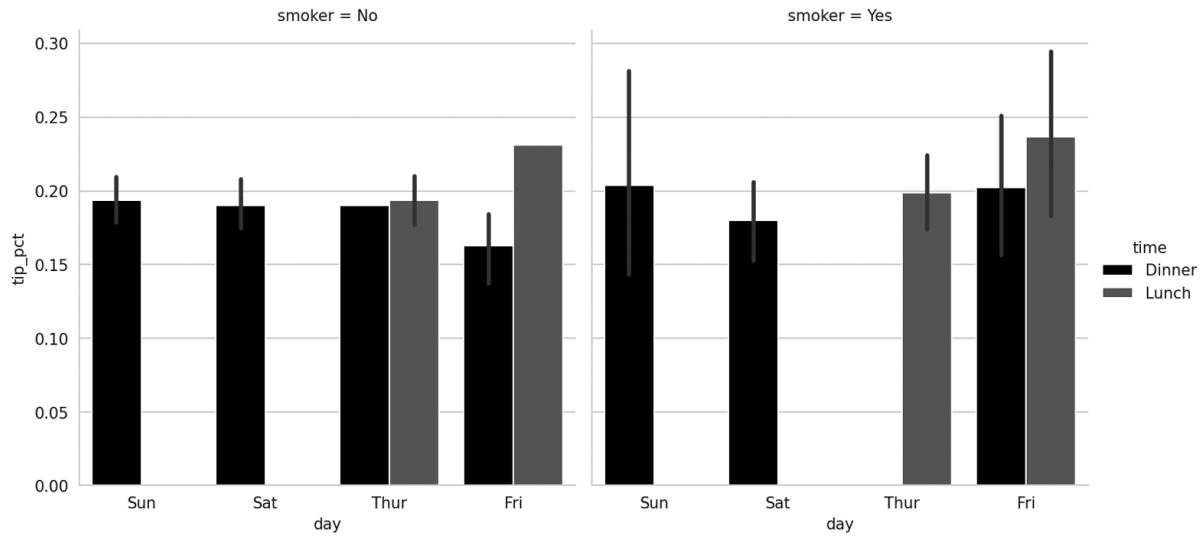


Figura 9.26. Porcentaje de propinas por día/hora/fumador.

En lugar de agrupar por “time” según distintos colores de barra dentro de una faceta, también podemos expandir la cuadrícula de facetas añadiendo una fila por valor `time` (véase la figura 9.27):

```
In [113]: sns.catplot(x="day", y="tip_pct", row="time",
.....:     col="smoker",
.....:     kind="bar", data=tips[tips.tip_pct < 1])
```

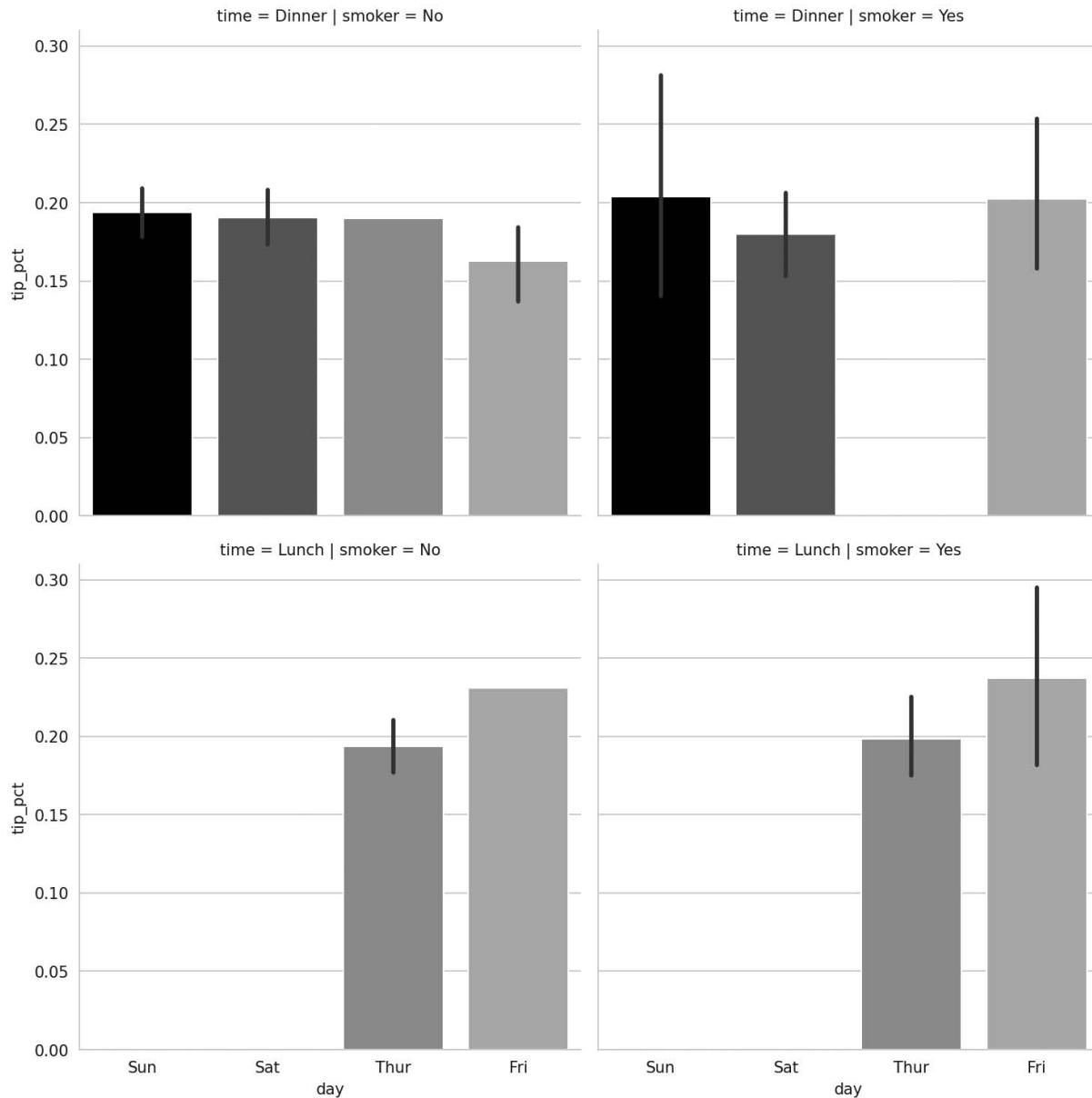


Figura 9.27. Porcentaje de propinas por día dividido por hora/fumador.

catplot soporta otros tipos de gráficos que pueden ser útiles dependiendo de lo que se esté tratando de mostrar. Por ejemplo, los diagramas de cajas (que muestran la mediana, los cuartiles y los valores atípicos) pueden ser un tipo de visualización eficaz (figura 9.28):

```
In [114]: sns.catplot(x="tip_pct", y="day", kind="box",
.....:         data=tips[tips.tip_pct < 0.5])
```

Es posible crear gráficos propios de cuadrícula de facetas utilizando la clase más general `seaborn.FacetGrid`. En la documentación de seaborn (<https://seaborn.pydata.org/>) se puede encontrar más información.

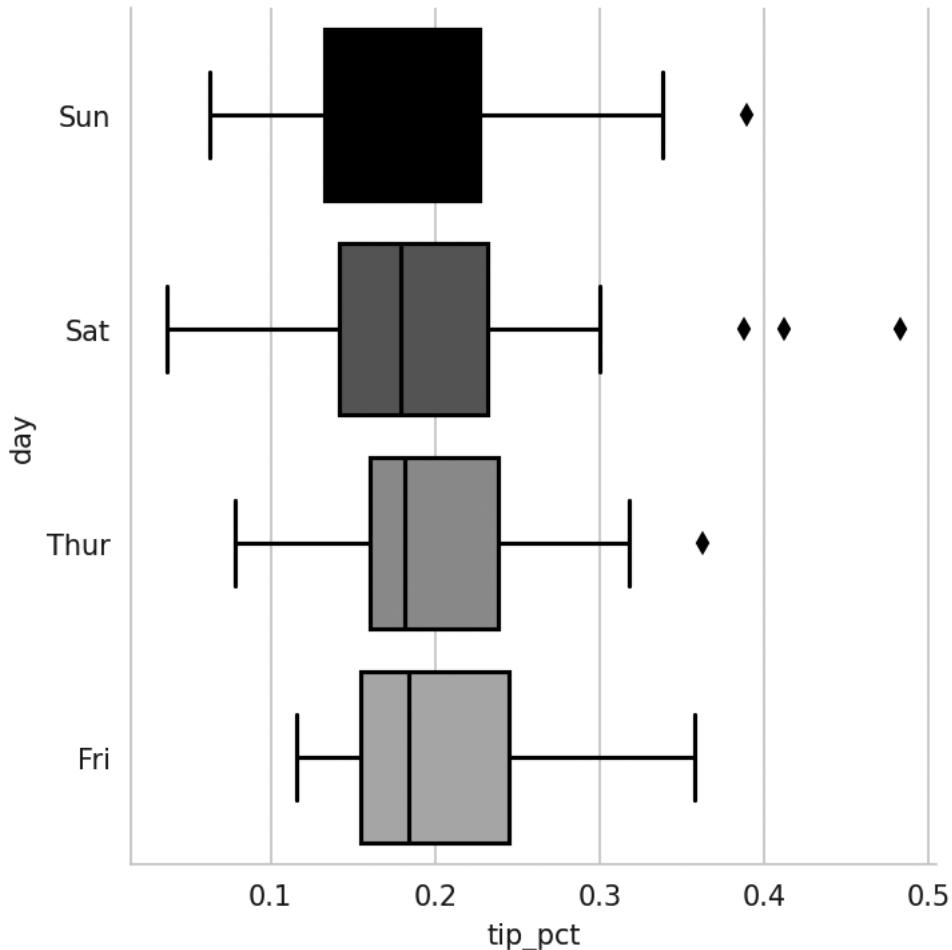


Figura 9.28. Diagrama de cajas del porcentaje de propinas por día.

9.3. Otras herramientas de visualización de Python

Como es habitual en código abierto, hay muchas opciones para crear gráficos en Python (demasiadas para listarlas aquí). Desde 2010, se han centrado muchos esfuerzos de desarrollo en crear gráficos interactivos para su publicación en la web. Con herramientas como Altair (<https://altair-viz.github.io/>), Bokeh (<http://bokeh.pydata.org>) y Plotly

(<https://plotly.com/python>), ahora es posible especificar gráficos dinámicos e interactivos en Python, destinados a su uso con navegadores web. Para crear gráficos estáticos para su impresión o para la web, recomiendo utilizar matplotlib y librerías basadas en matplotlib, como pandas y seaborn. Para otros requisitos de visualización de datos, puede resultar útil aprender cómo utilizar otra de las herramientas disponibles. Animo a los lectores a explorar el ecosistema, pues está en continua evolución y no deja de ofrecer innovaciones.

Un libro excelente sobre visualización de datos es *Fundamentals of Data Visualization*, de Claus O. Wilke (O'Reilly), disponible en papel o en el sitio web de Claus en <https://clauswilke.com/dataviz>.

9.4 Conclusión

El objetivo de este capítulo era iniciar a mis lectores en la visualización de datos básica mediante pandas, matplotlib y seaborn. Si comunicar visualmente los resultados del análisis de datos es importante en su trabajo, les animo a salir en busca de recursos para aprender más sobre la visualización efectiva de los datos. Es un campo de investigación activo, y permite practicar con muchos recursos de aprendizaje excelentes disponibles en línea y en papel.

En el siguiente capítulo nos centraremos en la agregación de datos y en las operaciones de grupo con pandas.

Agregación de datos y operaciones con grupos

Categorizar un conjunto de datos y aplicar una función a cada grupo, ya sea una agregación o una transformación, es un componente crítico de un flujo de trabajo de análisis de datos. Después de cargar, fusionar y preparar un conjunto de datos quizá necesitemos calcular estadísticas de grupo o tablas dinámicas, para posteriormente crear informes o visualizar los datos. Para ello pandas ofrece una versátil interfaz `groupby`, que permite segmentar y resumir conjuntos de datos de una forma lógica.

Una de las razones de la popularidad de las bases de datos relacionales y SQL (*Structured Query Language*: lenguaje de consulta estructurado) es la facilidad con la que los datos se pueden unir, filtrar, transformar y agregar. No obstante, lenguajes de consulta como SQL imponen ciertas limitaciones en los tipos de operaciones de grupos que se pueden realizar. Como veremos, con la expresividad que demuestran Python y pandas podemos realizar operaciones con grupos bastante complejas, expresándolas como funciones Python personalizadas que manipulan los datos asociados a cada grupo. En este capítulo aprenderemos los siguientes procesos:

- Dividir un objeto pandas en partes utilizando una o varias claves (en forma de funciones, arrays o nombres de columna de un dataframe).
- Calcular estadísticas de resumen de grupos, como recuento, promedio o desviación estándar, o una función definida por el usuario.
- Aplicar dentro del grupo transformaciones u otro tipo de manipulación, como normalización, regresión lineal, rango o selección de subconjuntos.
- Calcular tablas dinámicas y tabulaciones cruzadas.
- Realizar análisis de cuantiles y otros análisis estadísticos de grupo.



En este libro, a la agregación de datos de series temporales basada en el tiempo, un caso de uso especial de `groupby`, se le denomina remuestreo, y recibirá un tratamiento especial en el capítulo 11.

Igual que en el resto de los capítulos, empezamos importando NumPy y pandas:

In [12]: `import numpy as np`

In [13]: `import pandas as pd`

10.1 Entender las operaciones de grupos

Hadley Wickham, autor de muchos paquetes conocidos para el lenguaje de programación R, acuñó el término *split-apply-combine* (dividir-aplicar-combinar) para describir las operaciones de grupos. En la primera etapa del proceso, los datos contenidos en un objeto pandas, ya sea una serie, un dataframe o cualquier otro, se dividen en grupos basados en una o varias claves proporcionadas por el usuario. La división se realiza en un determinado eje del objeto. Por ejemplo, un dataframe puede agruparse por sus filas (`axis="index"`) o por sus columnas (`axis="columns"`). Una vez hecho esto, se aplica una función a cada grupo, para producir un nuevo valor. Por último, los resultados de todas esas aplicaciones de funciones se combinan en un objeto final. La forma de dicho objeto suele depender de lo que se les esté haciendo a los datos. La figura 10.1 muestra un modelo de una sencilla agregación de grupos.

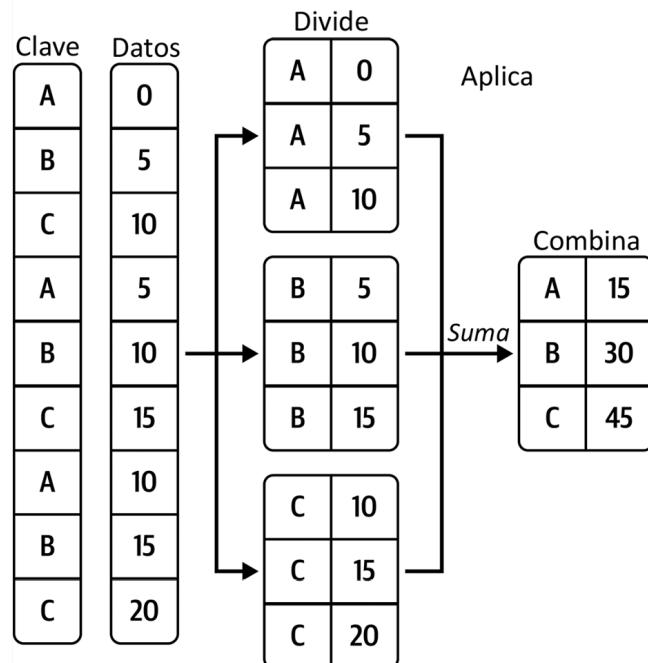


Figura 10.1. Ilustración de una agregación de grupos.

Las claves de agrupamiento pueden tomar muchas formas, no teniendo por qué ser todas del mismo tipo, pudiendo ser:

- Una lista o array de valores que tiene la misma longitud que el eje que está siendo agrupado.
- Un valor indicando un nombre de columna de un dataframe.
- Un diccionario o una serie que da una correspondencia entre los valores del eje que se agrupa y los nombres de los grupos.

- Una función que se invoca en el índice del eje o en las etiquetas individuales del índice.

Los últimos tres métodos son atajos para producir un array de valores que se utilizarán para dividir el objeto. No debemos preocuparnos si esto nos parece abstracto. A lo largo de este capítulo daré muchos ejemplos de todos estos métodos. Para empezar, aquí tenemos un pequeño conjunto de datos tabulares en forma de dataframe:

```
In [14]: df = pd.DataFrame({"key1" : ["a", "a", None, "b", "b", "a", None],
....: "key2" : pd.Series([1, 2, 1, 2, 1, None, 1], dtype="Int64"),
....: "data1" : np.random.standard_normal(7),
....: "data2" : np.random.standard_normal(7)})
```

```
In [15]: df
Out[15]:
```

	key1	key2	data1	data2
0	a	1	-0.204708	0.281746
1	a	2	0.478943	0.769023
2	None	1	-0.519439	1.246435
3	b	2	-0.555730	1.007189
4	b	1	1.965781	-1.296221
5	a	<NA>	1.393406	0.274992
6	None	1	0.092908	0.228913

Supongamos que queremos calcular la media de la columna data1 utilizando las etiquetas de key1. Hay varias formas de hacer esto. Una de ellas es acceder a data1 y llamar a groupby con la columna (una serie) en key1:

```
In [16]: grouped = df["data1"].groupby(df["key1"])
In [17]: grouped
Out[17]: <pandas.core.groupby.generic.SeriesGroupBy object at 0x7fa9270e0a00>
```

La variable grouped es ahora un objeto especial “GroupBy”. En realidad, no ha calculado nada todavía, excepto ciertos datos intermedios sobre la clave de grupo df["key1"]. La idea es que este objeto tenga toda la información necesaria para

aplicar después una determinada operación a cada uno de los grupos. Por ejemplo, para calcular las medias de los grupos podemos llamar al método `mean` de `GroupBy`:

```
In [18]: grouped.mean()
Out[18]:
key1

a          0.555881
b          0.705025

Name: data1, dtype: float64
```

Más tarde, en la sección 10.2 «Agregación de datos», explicaré más sobre lo que ocurre cuando se llama a `.mean()`. Lo importante aquí es que los datos (un objeto `Series`) se han agregado dividiéndolos según la clave de grupo, produciendo una nueva serie ahora indexada por los valores únicos de la columna `key1`. El índice del resultado tiene el nombre “`key1`” porque es el que tenía la columna `df[“key1”]` del `dataframe`.

Si en lugar de ello le hubiéramos pasado varios arrays como una lista, habríamos obtenido algo distinto:

```
In [19]: means = df[“data1”].groupby([df[“key1”], df[“key2”]]).mean()

In [20]: means
Out[20]:
key1      key2
a         1           -0.204708
          2           0.478943
b         1           1.965781
          2           -0.555730

Name: data1, dtype: float64
```

Aquí agrupamos los datos utilizando dos claves, y la serie resultante tiene ahora un índice jerárquico formado por los pares de claves únicas observados:

```
In [21]: means.unstack()
Out[21]:
key2      1           2
key1
a         -0.204708   0.478943
b         1.965781   -0.555730
```

En este ejemplo, las claves de los grupos son todas objetos Series, aunque podrían ser arrays cualesquiera de la longitud correcta:

```
In [22]: states = np.array(["OH", "CA", "CA", "OH", "OH", "CA", "OH"])
```

```
In [23]: years = [2005, 2005, 2006, 2005, 2006, 2005, 2006]
```

```
In [24]: df["data1"].groupby([states, years]).mean()  
Out[24]:
```

CA	2005	0.936175
	2006	-0.519439
OH	2005	-0.380219
	2006	1.029344

```
Name: data1, dtype: float64
```

Muchas veces la información de agrupamiento se encuentra en el mismo dataframe que los datos con los que se desea trabajar. En ese caso, se pueden pasar nombres de columna (ya sean cadenas de texto, números u otros objetos Python) como claves para los grupos:

```
In [25]: df.groupby("key1").mean()  
Out[25]:
```

	key2	data1	data2
key1			
a	1.5	0.555881	0.441920
b	1.5	0.705025	-0.144516

```
In [26]: df.groupby("key2").mean()  
Out[26]:
```

	data1	data2
key2		
1	0.333636	0.115218
2	-0.038393	0.888106

```
In [27]: df.groupby(["key1", "key2"]).mean()  
Out[27]:
```

	data1	data2
--	-------	-------

key1	key2		
a	1	-0.204708	0.281746
	2	0.478943	0.769023
b	1	1.965781	-1.296221
	2	-0.555730	1.007189

En el segundo caso, `df.groupby("key2").mean()`, se puede observar que no hay columna `key1` en el resultado. Como `df["key1"]` no contiene datos numéricos, se dice que es una columna «molesta», por lo que se excluye automáticamente del resultado. De forma predeterminada se agregan todas las columnas numéricas, aunque es posible filtrarlas en un subconjunto, como veremos pronto. Sin tener en cuenta el objetivo que tengamos al utilizar `groupby`, un método de `GroupBy` que se utiliza en general es `size`, que devuelve una serie formada por los tamaños de los grupos:

```
In [28]: df.groupby(["key1", "key2"]).size()
Out[28]:
```

key1	key2	
a	1	1
	2	1
b	1	1
	2	1

dtype: int64

Se puede observar que los valores faltantes de una clave de grupo quedan excluidos del resultado de forma predeterminada. Este comportamiento se puede deshabilitar pasándole `dropna=False` a `groupby`:

```
In [29]: df.groupby("key1", dropna=False).size()
Out[29]:
```

key1	
a	3
b	2
NaN	2
dtype:	int64

```
In [30]: df.groupby(["key1", "key2"], dropna=False).size()
Out[30]:
```

key1	key2	
------	------	--

```

a              1              1
              2              1
              <NA> 1
b              1              1
              2              1
NaN            1              2
dtype: int64

```

Una función de grupo muy parecida a size es count, que calcula el número de valores no nulos de cada grupo:

```
In [31]: df.groupby("key1").count()
Out[31]:
```

	key2	data1	data2
key1			
a	2	3	3
b	2	2	2

Iteración a través de grupos

El objeto devuelto por groupby soporta iteración, generando una secuencia de tuplas de 2 que contienen el nombre de grupo, además de los datos correspondientes. Consideremos lo siguiente:

```
In [32]: for name, group in df.groupby("key1"):
```

```

....:             print(name)
....:             print(group)
....:

a
    key1      key2      data1      data2
0     a        1   -0.204708   0.281746
1     a        2    0.478943   0.769023
5     a    <NA>    1.393406   0.274992
b
    key1      key2      data1      data2
3     b        2   -0.555730   1.007189
4     b        1    1.965781  -1.296221

```

En el caso de tener múltiples claves, el primer elemento de la tupla será una tupla de valores de clave:

```
In [33]: for (k1, k2), group in df.groupby(["key1", "key2"]):  
....:  
....:     print((k1, k2))  
....:     print(group)  
....:  
  
('a', 1)  
    key1      key2      data1      data2  
0       a        1   -0.204708  0.281746  
('a', 2)  
key1      key2      data1      data2  
1       a        2    0.478943  0.769023  
('b', 1)  
key1      key2      data1      data2  
4       b        1    1.965781 -1.296221  
('b', 2)  
key1      key2      data1      data2  
3       b        2   -0.55573  1.007189
```

Por supuesto, se puede elegir hacer lo que se quiera con los fragmentos de datos. Por ejemplo, puede resultar útil calcular un diccionario de dichos datos como un solo elemento:

```
In [34]: pieces = {name: group for name, group in df.groupby("key1")}
```

```
In [35]: pieces["b"]
```

	key1	key2	data1	data2
3	b	2	-0.555730	1.007189
4	b	1	1.965781	-1.296221

Por defecto, `groupby` agrupa según `axis="index"`, pero se puede agrupar según cualquiera de los otros ejes. Por ejemplo, podríamos agrupar aquí las columnas de nuestro ejemplo `df`, ya empiecen por “key” o por “data”:

```
In [36]: grouped = df.groupby({"key1": "key", "key2": "key",
...:     "data1": "data", "data2": "data"}, axis="columns")
```

Podemos visualizar los grupos así:

```
In [37]: for group_key, group_values in grouped:
```

```
....:         print(group_key)
....:         print(group_values)
....:
```

```
data
```

	data1	data2
0	-0.204708	0.281746
1	0.478943	0.769023
2	-0.519439	1.246435
3	-0.555730	1.007189
4	1.965781	-1.296221
5	1.393406	0.274992
6	0.092908	0.228913

```
key
```

	key1	key2
0	a	1
1	a	2
2	None	1
3	b	2
4	b	1
5	a	<NA>
6	None	1

Selección de una columna o subconjunto de columnas

El efecto de indexar un objeto GroupBy creado a partir de un dataframe con un nombre de columna o un array de nombres de columnas es que se crean subconjuntos de dichas columnas para su agregación. Esto significa que estas dos líneas de código:

```
df.groupby("key1")["data1"]
```

```
df.groupby("key1")[["data2"]]
```

Son muy prácticas para:

```
df["data1"].groupby(df["key1"])
```

```
df[["data2"]].groupby(df["key1"])
```

Especialmente con grandes conjuntos de datos, puede interesar agregar solamente unas cuantas columnas. Por ejemplo, en el conjunto de datos anterior, para calcular la media solo para la columna data2 y obtener el resultado como un dataframe, podríamos escribir:

```
In [38]: df.groupby(["key1", "key2"])["data2"].mean()  
Out[38]:
```

key1	key2	data2
a	1	0.281746
	2	0.769023
b	1	-1.296221
	2	1.007189

El objeto devuelto por esta operación de indexado es un dataframe agrupado si se pasa una lista o array, o una serie agrupada si solo se pasa un nombre de columna como un escalar:

```
In [39]: s_grouped = df.groupby(["key1", "key2"])["data2"]  
  
In [40]: s_grouped  
Out[40]: <pandas.core.groupby.generic.SeriesGroupBy object at 0x7fa9270e3520>  
  
In [41]: s_grouped.mean()  
Out[41]:
```

key1	key2	data2
a	1	0.281746
	2	0.769023
b	1	-1.296221
	2	1.007189

Name: data2, dtype: float64

Agrupamiento con diccionarios y series

La información de agrupamiento puede existir en una forma distinta a un array. Veamos otro dataframe de ejemplo:

```
In [42]: people = pd.DataFrame(np.random.standard_normal((5, 5)),
```

```
....:     columns=["a", "b", "c", "d", "e"],  
....:     index=["Joe", "Steve", "Wanda", "Jill", "Trey"])
```

```
In [43]: people.iloc[2:3, [1, 2]] = np.nan # Añade algunos valores NA
```

```
In [44]: people
```

```
Out[44]:
```

	a	b	c	d	e
Joe	1.352917	0.886429	-2.001637	-0.371843	1.669025
Steve	-0.438570	-0.539741	0.476985	3.248944	-1.021228
Wanda	-0.577087	Nan	NaN	0.523772	0.000940
Jill	1.343810	-0.713544	-0.831154	-2.370232	-1.860761
Trey	-0.860757	0.560145	-1.265934	0.119827	-1.063512

Supongamos ahora que tengo una correspondencia de grupos para las columnas y quiero sumar las columnas por grupos:

```
In [45]: mapping = {"a": "red", "b": "red", "c": "blue",  
....:     "d": "blue", "e": "red", "f": "orange"}
```

Podríamos construir entonces un array a partir de este diccionario para pasarlo a groupby, pero en lugar de ello podemos simplemente pasar el diccionario (incluí la clave “f” para resaltar que las claves de agrupamiento no utilizadas son correctas):

```
In [46]: by_column = people.groupby(mapping, axis="columns")
```

```
In [47]: by_column.sum()  
Out[47]:
```

	blue	red
Joe	-2.373480	3.908371
Steve	3.725929	-1.999539
Wanda	0.523772	-0.576147
Jill	-3.201385	-1.230495
Trey	-1.146107	-1.364125

La misma funcionalidad aplica a las series, que se pueden ver como un mapeado de tamaño fijo:

```
In [48]: map_series = pd.Series(mapping)
```

```
In [49]: map_series  
Out[49]:
```

```
a          red  
b          red  
c        blue  
d        blue  
e          red  
f    orange
```

dtype: object

```
In [50]: people.groupby(map_series, axis="columns").count()  
Out[50]:
```

	blue	red
Joe	2	3
Steve	2	3
Wanda	1	2
Jill	2	3
Trey	2	3

Agrupamiento con funciones

Utilizar funciones de Python es una forma más genérica de definir un mapeado de grupos en comparación con un diccionario o un objeto Series. Cualquier función pasada como clave de grupo será llamada una vez por valor de índice (o una vez por valor de columna si se utiliza `axis="columns"`), utilizando los valores devueltos como nombres de los grupos. Más concretamente, consideremos el dataframe de ejemplo de la sección anterior, que tiene nombres de pila de personas como valores de índice. Supongamos que queremos agrupar según la longitud del nombre. Aunque podríamos calcular un array de longitudes de cadenas de texto, es más sencillo pasar simplemente la función `len`:

```
In [51]: people.groupby(len).sum()  
Out[51]:
```

	a	b	c	d	e
3	1.352917	0.886429	-2.001637	-0.371843	1.669025
4	0.483052	-0.153399	-2.097088	-2.250405	-2.924273
5	-1.015657	-0.539741	0.476985	3.772716	-1.020287

Mezclar funciones con arrays, diccionarios u objetos Series no es un problema, ya que internamente todo se convierte en arrays:

```
In [52]: key_list = ["one", "one", "one", "two", "two"]
```

```
In [53]: people.groupby([len, key_list]).min()
```

```
Out[53]:
```

	a	b	c	d	e
3	one	1.352917	0.886429	-2.001637	-0.371843
4	two	-0.860757	-0.713544	-1.265934	-2.370232
5	one	-0.577087	-0.539741	0.476985	0.523772

Agrupamiento por niveles de índice

Una última ventaja de los conjuntos de datos jerárquicamente indexados es la capacidad para agregar utilizando uno de los niveles de un índice de ejes. Veamos un ejemplo:

```
In [54]: columns = pd.MultiIndex.from_arrays([[ "US", "US", "US",  
"JP", "JP"],
```

```
.....: [1, 3, 5, 1, 3]],  
.....: names=[ "cty", "tenor"])
```

```
In [55]: hier_df = pd.DataFrame(np.random.standard_normal((4, 5)),  
columns=columns)
```

```
In [56]: hier_df
```

```
Out[56]:
```

	US			JP	
cty	1	3	5	1	3
tenor	0.332883	-2.359419	-0.199543	-1.541996	-0.970736
0	-1.307030	0.286350	0.377984	-0.753887	0.331286
1	1.349742	0.069877	0.246674	-0.011862	1.004812
2	1.327195	-0.919262	-1.549106	0.022185	0.758363

Para agrupar por nivel, pasamos el número o el nombre del nivel utilizando la palabra clave `level`:

```
In [57]: hier_df.groupby(level="cty", axis="columns").count()
```

```
Out[57]:
```

	JP	US
0	2	3
1	2	3
2	2	3
3	2	3

10.2 Agregación de datos

Las agregaciones hacen referencia a cualquier transformación de datos que produce valores escalares a partir de arrays. Los ejemplos anteriores han utilizado varias, incluyendo `mean`, `count`, `min` y `sum`. Quizá alguno de mis lectores se esté preguntando qué pasa cuando se invoca a `mean()` sobre un objeto `GroupBy`. Muchas agregaciones habituales, como las que resume la tabla 10.1, tienen implementaciones optimizadas. Sin embargo, no estamos limitados solo a este conjunto de métodos.

Tabla 10.1. Métodos groupby optimizados.

Nombre de función	Descripción
<code>any</code> , <code>all</code>	Devuelve <code>True</code> si alguno (uno o varios) o todos los valores que no son <code>NA</code> evalúan a <code>True</code> .
<code>count</code>	Número de valores que no son <code>NA</code> .
<code>cummin</code> , <code>cummax</code>	Mínimo y máximo acumulativo de valores que no son <code>NA</code> .
<code>cumsum</code>	Suma acumulativa de valores que no son <code>NA</code> .
<code>cumprod</code>	Producto acumulativo de valores que no son <code>NA</code> .
<code>first</code> , <code>last</code>	Primer y último valor que no es <code>NA</code> .
<code>mean</code>	Media de los valores que no son <code>NA</code> .
<code>median</code>	Media aritmética de los valores que no son <code>NA</code> .
<code>min</code> , <code>max</code>	Mínimo y máximo de los valores que no son <code>NA</code> .
<code>nth</code>	Recupera el valor que aparecería en la posición <code>n</code> (enésima) con los datos ordenados.
<code>ohlc</code>	Calcula cuatro estadísticas " <i>Open-High-Low-Close</i> " (abrir-alto-bajo-cerrar) para datos de tipo serie temporal.
<code>prod</code>	Producto de los valores que no son <code>NA</code> .
<code>quantile</code>	Calcula el cuantil de la muestra.
<code>rank</code>	Rangos ordinales de valores que no son <code>NA</code> , igual que llamar a <code>Series.rank</code> .

Nombre de función	Descripción
size	Calcula tamaños de grupo, devolviendo el resultado como una serie.
sum	Suma de los valores que no son NA.
std, var	Desviación estándar y varianza de la muestra.

Podemos utilizar las agregaciones que se nos ocurran y llamar, además, a cualquier método que también esté definido en el objeto que está siendo agrupado. Por ejemplo, el método `nsmallest` del objeto Series selecciona de los datos el número mínimo requerido de valores. Aunque GroupBy no implementa de manera explícita este método, aún podemos usarlo con una implementación no optimizada. Internamente, GroupBy segmenta la serie, llama a `piece.nsmallest(n)` para cada fragmento y después monta esos resultados en el objeto final:

```
In [58]: df
Out[58]:
```

	key1	key2	data1	data2
0	a	1	-0.204708	0.281746
1	a	2	0.478943	0.769023
2	None	1	-0.519439	1.246435
3	b	2	-0.555730	1.007189
4	b	1	1.965781	-1.296221
5	a	<NA>	1.393406	0.274992
6	None	1	0.092908	0.228913

```
In [59]: grouped = df.groupby("key1")
```

```
In [60]: grouped["data1"].nsmallest(2)
Out[60]:
```

key1		
a	0	-0.204708
	1	0.478943
b	3	-0.555730
	4	1.965781

```
Name: data1, dtype: float64
```

Para utilizar nuestras propias funciones de agregación, pasamos cualquier función que agregue un array al método `aggregate` o a su alias abreviado `agg`:

```
In [61]: def peak_to_peak(arr):
....:
    return arr.max()-arr.min()
```

```
In [62]: grouped.agg(peak_to_peak)
Out[62]:
```

	key2	data1	data2
key1			
a	1	1.598113	0.494031
b	1	2.521511	2.303410

Algunos métodos, como describe, también funcionan, incluso aunque no sean agregaciones estrictamente hablando:

```
In [63]: grouped.describe()
Out[63]:
```

	key2	data1	...\\
	count mean std min 25% 50% 75% max count mean ...		
key1 ...			
a	2.0 1.5 0.707107 1.0 1.25 1.5 1.75 2.0 3.0 0.555881 ...		
b	2.0 1.5 0.707107 1.0 1.25 1.5 1.75 2.0 2.0 0.705025 ...		
	data2 \\		
	75% max count mean std min 25%		
key1			
a	0.936175 1.393406 3.0 0.441920 0.283299 0.274992 0.278369		
b	1.335403 1.965781 2.0 -0.144516 1.628757 -1.296221 -0.720368		
	50% 75% max		
key1			
a	0.281746 0.525384 0.769023		
b	-0.144516 0.431337 1.007189		

[2 rows x 24 columns]

Explicaré con más detalle lo que ha ocurrido aquí en la sección posterior «El método apply: un *split-apply-combine* general».

Las funciones de agregación personalizadas suelen ser mucho más lentas que las optimizadas que se pueden



encontrar en la tabla 10.1. Esto se debe a que existe una cierta sobrecarga adicional (llamadas a funciones, reordenación de datos) en la construcción de los fragmentos intermedios de los datos de los grupos.

Aplicación de varias funciones a columnas

Volvamos al conjunto de datos de propinas empleado en el capítulo anterior. Tras cargarlo con pandas.read_csv, añadimos una columna para el porcentaje de propinas:

```
In [64]: tips = pd.read_csv("examples/tips.csv")
```

```
In [65]: tips.head()
```

```
Out[65]:
```

	total_bill	tip	smoker	day	time	size
0	16.99	1.01	No	Sun	Dinner	2
1	10.34	1.66	No	Sun	Dinner	3
2	21.01	3.50	No	Sun	Dinner	3
3	23.68	3.31	No	Sun	Dinner	2
4	24.59	3.61	No	Sun	Dinner	4

Añadimos la columna tip_pct que muestra el porcentaje que supone la propina en la factura total:

```
In [66]: tips["tip_pct"] = tips["tip"] / tips["total_bill"]
```

```
In [67]: tips.head()
```

```
Out[67]:
```

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.059447
1	10.34	1.66	No	Sun	Dinner	3	0.160542
2	21.01	3.50	No	Sun	Dinner	3	0.166587
3	23.68	3.31	No	Sun	Dinner	2	0.139780
4	24.59	3.61	No	Sun	Dinner	4	0.146808

Como ya hemos visto, agregar una serie o todas las columnas de un dataframe es simplemente cuestión de usar aggregate (o agg) con la función deseada o llamar a un método como mean o std. Sin embargo, quizá sea preferible realizar este proceso utilizando otra función, dependiendo de la columna, o varias funciones al mismo tiempo. Por suerte, esto es posible, y lo ilustraré con unos cuantos ejemplos. Primero, agruparé las propinas por day y smoker:

```
In [68]: grouped = tips.groupby(["day", "smoker"])
```

Para estadísticas descriptivas como las de la tabla 10.1, se puede pasar el nombre de la función como una cadena de texto:

```
In [69]: grouped_pct = grouped["tip_pct"]
```

```
In [70]: grouped_pct.agg("mean")
```

```
Out[70]:
```

day	smoker	
Fri	No	0.151650
	Yes	0.174783
Sat	No	0.158048
	Yes	0.147906
Sun	No	0.160113
	Yes	0.187250
Thur	No	0.160298
	Yes	0.163863

```
Name: tip_pct, dtype: float64
```

Si en lugar de esto pasamos una lista de funciones o de nombres de funciones, obtenemos de vuelta un dataframe en el que los nombres de las columnas se han tomado de las funciones:

```
In [71]: grouped_pct.agg(["mean", "std", "peak_to_peak"])
```

```
Out[71]:
```

day	smoker	mean	std	peak_to_peak
Fri	No	0.151650	0.028123	0.067349
	Yes	0.174783	0.051293	0.159925
Sat	No	0.158048	0.039767	0.235193
	Yes	0.147906	0.061375	0.290095
Sun	No	0.160113	0.042347	0.193226
	Yes	0.187250	0.154134	0.644685
Thur	No	0.160298	0.038774	0.193350
	Yes	0.163863	0.039389	0.151240

Aquí le pasamos a agg una lista de funciones de agregación para evaluar de forma independiente según los grupos de datos.

No es necesario aceptar los nombres que GroupBy da a las columnas, ya que las funciones lambda llevan el nombre "<lambda>", lo que las hace difíciles de identificar (se puede comprobar mirando el atributo `_name_` de una función). De esta forma, si

se pasa una lista de tuplas (name, function), el primer elemento de cada tupla se utilizará para los nombres de las columnas del dataframe (se puede pensar en una lista de tuplas de 2 como en un mapeado ordenado):

```
In [72]: grouped_pct.agg([('average', "mean"), ("stdev", np.std)])
Out[72]:
```

			average	stdev
day	smoker			
Fri	No		0.151650	0.028123
	Yes		0.174783	0.051293
Sat	No		0.158048	0.039767
	Yes		0.147906	0.061375
Sun	No		0.160113	0.042347
	Yes		0.187250	0.154134
Thur	No		0.160298	0.038774
	Yes		0.163863	0.039389

Con un dataframe tenemos más opciones, porque es posible especificar la lista de funciones que se van a aplicar a todas las columnas o distintas funciones por columna. Para empezar, supongamos que queremos calcular las mismas tres estadísticas para las columnas tip_pct y total_bill:

```
In [73]: functions = ["count", "mean", "max"]
```

```
In [74]: result = grouped[['tip_pct', 'total_bill']].agg(functions)
```

```
In [75]: result
```

```
Out[75]:
```

day	smoker	tip_pct		total_bill		
		count	mean	max	count	mean
Fri	No	4	0.151650	0.187735	4	18.420000
	Yes	15	0.174783	0.263480	15	40.17
Sat	No	45	0.158048	0.291990	45	48.33
	Yes	42	0.147906	0.325733	42	50.81
Sun	No	57	0.160113	0.252672	57	48.17
	Yes	19	0.187250	0.710345	19	45.35
Thur	No	45	0.160298	0.266312	45	41.19
	Yes	17	0.163863	0.241255	17	43.11

Como se puede observar, el dataframe resultante tiene columnas jerárquicas. Similar resultado se obtendría agregando cada columna por separado y utilizando concat para unir los resultados empleando los nombres de columna como argumento de keys:

```
In [76]: result["tip_pct"]
Out[76]:
```

			count	mean	max
day	smoker				
Fri	No		4	0.151650	0.187735
	Yes		15	0.174783	0.263480
Sat	No		45	0.158048	0.291990
	Yes		42	0.147906	0.325733
Sun	No		57	0.160113	0.252672
	Yes		19	0.187250	0.710345
Thur	No		45	0.160298	0.266312
	Yes		17	0.163863	0.241255

Igual que antes, podemos pasar una lista de tuplas con nombres personalizados:

```
In [77]: ftuples = [("Average", "mean"), ("Variance", np.var)]
In [78]: grouped[["tip_pct", "total_bill"]].agg(ftuples)
Out[78]:
```

		tip_pct		total_bill	
day	smoker	Average	Variance	Average	Variance
Fri	No	0.151650	0.000791	18.420000	25.596333
	Yes	0.174783	0.002631	16.813333	82.562438
Sat	No	0.158048	0.001581	19.661778	79.908965
	Yes	0.147906	0.003767	21.276667	101.387535
Sun	No	0.160113	0.001793	20.506667	66.099980
	Yes	0.187250	0.023757	24.120000	109.046044
Thur	No	0.160298	0.001503	17.113111	59.625081
	Yes	0.163863	0.001551	19.190588	69.808518

Supongamos ahora que queremos aplicar funciones en principio distintas a una columna o a varias. Para ello le pasamos a agg un diccionario que contiene un mapeado de nombres de columna asignados a cualquiera de las especificaciones de función listadas hasta ahora:

```
In [79]: grouped.agg({"tip" : np.max, "size" : "sum"})
Out[79]:
```

day	smoker	tip	size
Fri	No	3.50	9
	Yes	4.73	31
Sat	No	9.00	115
	Yes	10.00	104
Sun	No	6.00	167
	Yes	6.50	49
Thur	No	6.70	112
	Yes	5.00	40

```
In [80]: grouped.agg({"tip_pct" : ["min", "max", "mean", "std"],
....: "size" : "sum"})
Out[80]:
```

day	smoker	tip_pct			size	
		min	max	mean	std	sum
Fri	No	0.120385	0.187735	0.151650	0.028123	9
	Yes	0.103555	0.263480	0.174783	0.051293	31
Sat	No	0.056797	0.291990	0.158048	0.039767	115
	Yes	0.035638	0.325733	0.147906	0.061375	104
Sun	No	0.059447	0.252672	0.160113	0.042347	167
	Yes	0.065660	0.710345	0.187250	0.154134	49
Thur	No	0.072961	0.266312	0.160298	0.038774	112
	Yes	0.090014	0.241255	0.163863	0.039389	40

Un dataframe tendrá columnas jerárquicas solo si se aplican varias funciones al menos a una columna.

Devolución de datos agregados sin índices de fila

En todos los ejemplos examinados hasta ahora, los datos agregados vuelven con un índice, posiblemente jerárquico, formado por las combinaciones de claves de grupos únicas. Como esto no siempre es deseable, se puede deshabilitar este comportamiento en la mayoría de los casos pasando `as_index=False` a `groupby`:

```
In [81]: tips.groupby(["day", "smoker"], as_index=False).mean()
Out[81]:
```

	day	smoker	total_bill	tip	size	tip_pct
0	Fri	No	18.420000	2.812500	2.250000	0.151650
1	Fri	Yes	16.813333	2.714000	2.066667	0.174783
2	Sat	No	19.661778	3.102889	2.555556	0.158048
3	Sat	Yes	21.276667	2.875476	2.476190	0.147906
4	Sun	No	20.506667	3.167895	2.929825	0.160113
5	Sun	Yes	24.120000	3.516842	2.578947	0.187250
6	Thur	No	17.113111	2.673778	2.488889	0.160298
7	Thur	Yes	19.190588	3.030000	2.352941	0.163863

Por supuesto, siempre es posible obtener el resultado en este formato llamando a `reset_index` sobre el resultado. Usar el argumento `as_index=False` evita ciertos cálculos innecesarios.

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

El método GroupBy más genérico es `apply`, justamente el tema de esta sección. El método `apply` divide el objeto que se está manipulando en partes, invoca la función pasada sobre cada una de ellas y después trata de concatenarlas.

Volviendo al conjunto de datos de propinas de antes, supongamos que queremos seleccionar los cinco valores `tip_pct` más importantes por grupo. Primero escribimos una función que seleccione las filas con los mayores valores en una determinada columna:

```
In [82]: def top(df, n=5, column="tip_pct"):

....:     return df.sort_values(column, ascending=False)[:n]
```

```
In [83]: top(tips, n=6)
Out[83]:
```

	total_bill	tip	smoker	day	time	size	tip_pct
172	7.25	5.15	Yes	Sun	Dinner	2	0.710345
178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
232	11.61	3.39	No	Sat	Dinner	2	0.291990
183	23.17	6.50	Yes	Sun	Dinner	4	0.280535
109	14.31	4.00	Yes	Sat	Dinner	2	0.279525

Si agrupamos ahora por `smoker`, por ejemplo, y llamamos a `apply` con esta función, obtenemos lo siguiente:

```
In [84]: tips.groupby("smoker").apply(top)
Out[84]:
```

		total_bill	tip	smoker	day	time	size	tip_pct
smoker								
No	232	11.61	3.39		No	Sat	Dinner	2 0.291990
	149	7.51	2.00		No	Thur	Lunch	2 0.266312
	51	10.29	2.60		No	Sun	Dinner	2 0.252672
	185	20.69	5.00		No	Sun	Dinner	5 0.241663
	88	24.71	5.85		No	Thur	Lunch	2 0.236746
Yes	172	7.25	5.15	Yes	Sun	Dinner	2	0.710345
	178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
	67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
	183	23.17	6.50	Yes	Sun	Dinner	4	0.280535
	109	14.31	4.00	Yes	Sat	Dinner	2	0.279525

¿Qué ha ocurrido aquí? Primero, el dataframe `tips` se ha dividido en grupos basándose en el valor de `smoker`. Después se llama a la función `top` en cada grupo, y el resultado de cada llamada a función se une utilizando `pandas.concat`, etiquetando las partes con los nombres de grupo. De ahí que el resultado tenga un índice jerárquico con un nivel interior que contiene los valores de índice del dataframe original.

Si se le pasa a `apply` una función que toma otros argumentos o palabras clave, se le pueden pasar después de la función:

```
In [85]: tips.groupby(["smoker", "day"]).apply(top, n=1,
column="total_bill")
Out[85]:
```

		total_bill	tip	smoker	day	time	size	tip_pct
smoker	day							
No	Fri 94	22.75	3.25		No	Fri	Dinner	2 0.142857
	Sat 212	48.33	9.00		No	Sat	Dinner	4 0.186220
	Sun 156	48.17	5.00		No	Sun	Dinner	6 0.103799
	Thur 142	41.19	5.00		No	Thur	Lunch	5 0.121389
Yes	Fri 95	40.17	4.73	Yes	Fri	Dinner	4	0.117750
	Sat 170	50.81	10.00	Yes	Sat	Dinner	3	0.196812
	Sun 182	45.35	3.50	Yes	Sun	Dinner	3	0.077178
	Thur 197	43.11	5.00	Yes	Thur	Lunch	4	0.115982

Más allá de esta mecánica de uso básica, sacar el máximo partido de apply puede requerir un poco de creatividad. Lo que ocurre dentro de la función pasada es cosa del usuario; debe devolver o bien un objeto pandas o un valor escalar. El resto de este capítulo consistirá principalmente en ejemplos que muestran cómo resolver varios problemas utilizando groupby.

Por ejemplo, anteriormente he llamado a describe en un objeto GroupBy:

```
In [86]: result = tips.groupby("smoker")["tip_pct"].describe()
```

```
In [87]: result
```

```
Out[87]:
```

	count	mean	std	min	25%	50%	75%	\
smoker								
No	151.0	0.159328	0.039910	0.056797	0.136906	0.155625	0.185014	
Yes	93.0	0.163196	0.085119	0.035638	0.106771	0.153846	0.195059	

	max
smoker	
No	0.291990
Yes	0.710345

```
In [88]: result.unstack("smoker")
```

```
Out[88]:
```

		smoker
count	No	151.000000
	Yes	93.000000
mean	No	0.159328
	Yes	0.163196
std	No	0.039910
	Yes	0.085119
min	No	0.056797
	Yes	0.035638
25%	No	0.136906
	Yes	0.106771
50%	No	0.155625
	Yes	0.153846
75%	No	0.185014
	Yes	0.195059
max	No	0.291990
	Yes	0.710345

```
dtype: float64
```

Dentro de GroupBy, al invocar un método como describe, en realidad es solo un atajo para:

```
def f(group):  
    return group.describe()  
  
grouped.apply(f)
```

Supresión de las claves de grupos

En los ejemplos anteriores podemos ver que el objeto resultante tiene un índice jerárquico formado a partir de las claves de grupos, además de los índices de cada parte del objeto original. Se puede deshabilitar esto pasándole group_keys=False a groupby:

```
In [89]: tips.groupby("smoker", group_keys=False).apply(top)  
Out[89]:
```

	total_bill	tip	smoker	day	time	size	tip_pct
232	11.61	3.39	No	Sat	Dinner	2	0.291990
149	7.51	2.00	No	Thur	Lunch	2	0.266312
51	10.29	2.60	No	Sun	Dinner	2	0.252672
185	20.69	5.00	No	Sun	Dinner	5	0.241663
88	24.71	5.85	No	Thur	Lunch	2	0.236746
172	7.25	5.15	Yes	Sun	Dinner	2	0.710345
178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
183	23.17	6.50	Yes	Sun	Dinner	4	0.280535
109	14.31	4.00	Yes	Sat	Dinner	2	0.279525

Análisis de cuantil y contenedor

Como quizá recuerden mis lectores del capítulo 8, pandas dispone de diversas herramientas, en particular pandas.cut y pandas.qcut, para dividir los datos en contenedores con los intervalos que cada uno elija, o mediante cuantiles de muestra. Combinar estas funciones con groupby resulta conveniente para realizar análisis de contenedor o cuantil en un conjunto de datos. Veamos un sencillo conjunto de datos aleatorio y una categorización de contenedor de la misma longitud utilizando pandas.cut:

```
In [90]: frame = pd.DataFrame({"data1":  
    np.random.standard_normal(1000),  
    ....: "data2": np.random.standard_normal(1000)})
```

```
In [91]: frame.head()  
Out[91]:
```

	data1	data2
0	-0.660524	-0.612905
1	0.862580	0.316447
2	-0.010032	0.838295
3	0.050009	-1.034423
4	0.670216	0.434304

```
In [92]: quartiles = pd.cut(frame["data1"], 4)
```

```
In [93]: quartiles.head(10)  
Out[93]:
```

0	(-1.23, 0.489]
1	(0.489, 2.208]
2	(-1.23, 0.489]
3	(-1.23, 0.489]
4	(0.489, 2.208]
5	(0.489, 2.208]
6	(-1.23, 0.489]
7	(-1.23, 0.489]
8	(-2.956, -1.23]
9	(-1.23, 0.489]

```
Name: data1, dtype: category
```

```
Categories (4, interval[float64, right]): [(-2.956, -1.23] < (-1.23, 0.489] < (0.489, 2.208] < (2.208, 3.928)]
```

El objeto Categorical devuelto por cut se le puede pasar directamente a groupby. Así, podríamos calcular un conjunto de estadísticas de grupo para los cuartiles, de este modo:

```
In [94]: def get_stats(group):
```

```
....:     return pd.DataFrame(  
....:         {"min": group.min(), "max": group.max(),  
....:          "count": group.count(), "mean": group.mean()}  
....:     )
```

```
In [95]: grouped = frame.groupby(quartiles)
```

```
In [96]: grouped.apply(get_stats)  
Out[96]:
```

			min	max	count	mean
data1						
(-2.956, -1.23]	data1	-2.949343	-1.230179	94	-1.658818	
(-1.23, 0.489]	data2	-3.399312	1.670835	94	-0.033333	
(0.489, 2.208]	data1	-1.228918	0.488675	598	-0.329524	
	data2	-2.989741	3.260383	598	-0.002622	
(2.208, 3.928]	data1	0.489965	2.200997	298	1.065727	
	data2	-3.745356	2.954439	298	0.078249	
	data1	2.212303	3.927528	10	2.644253	
	data2	-1.929776	1.765640	10	0.024750	

Recordemos que el mismo resultado se podría haber calculado de un modo más sencillo con:

```
In [97]: grouped.agg(["min", "max", "count", "mean"])
Out[97]:
```

	data1	data2	\				
	min	max	count	mean	min	max	count
data1							
(-2.956, -1.23]	-2.949343	-1.230179	94	-1.658818	-3.399312	1.670835	94
(-1.23, 0.489]	-1.228918	0.488675	598	-0.329524	-2.989741	3.260383	598
(0.489, 2.208]	0.489965	2.200997	298	1.065727	-3.745356	2.954439	298
(2.208, 3.928]	2.212303	3.927528	10	2.644253	-1.929776	1.765640	10
	mean						
data1							
(-2.956, -0.033333							
-1.23]							
(-1.23, -0.002622							
0.489]							
(0.489, 0.078249							
2.208]							
(2.208, 0.024750							

Estos eran contenedores de la misma longitud; para calcular contenedores del mismo tamaño basándonos en cuantiles de muestra, empleamos pandas.qcut. Podemos pasar 4 como número de cuartiles de muestra para el cálculo de

contenedores, y pasar `labels=False` para obtener solo los índices de cuartiles en lugar de intervalos:

```
In [98]: quartiles_samp = pd.qcut(frame["data1"], 4, labels=False)
```

```
In [99]: quartiles_samp.head()
```

```
Out[99]:
```

0	1
1	3
2	2
3	2
4	3

```
Name: data1, dtype: int64
```

```
In [100]: grouped = frame.groupby(quartiles_samp)
```

```
In [101]: grouped.apply(get_stats)
```

```
Out[101]:
```

		min	max	count	mean
data1					
0	data1	-2.949343	-0.685484	250	-1.212173
	data2	-3.399312	2.628441	250	-0.027045
1	data1	-0.683066	-0.030280	250	-0.368334
	data2	-2.630247	3.260383	250	-0.027845
2	data1	-0.027734	0.618965	250	0.295812
	data2	-3.056990	2.458842	250	0.014450
3	data1	0.623587	3.927528	250	1.248875
	data2	-3.745356	2.954439	250	0.115899

Ejemplo: Rellenar valores faltantes con valores específicos de grupo

Cuando estamos limpiando datos ausentes, en algunos casos podemos eliminar observaciones de datos utilizando `dropna`, pero en otros quizá sea preferible llenar los valores nulos (NA) empleando un valor fijo u otro valor derivado de los datos. La herramienta adecuada aquí es `fillna`; por ejemplo, aquí relleno los valores nulos con la media:

```
In [102]: s = pd.Series(np.random.standard_normal(6))
```

```
In [103]: s[::2] = np.nan
```

```
In [104]: s
```

```
Out[104]:
```

```

NaN
0          0.227290
1           NaN
2         -2.153545
3           NaN
4         -0.375842
5

dtype:float64

In [105]: s.fillna(s.mean())
Out[105]:


-0.767366
0          0.227290
1         -0.767366
2         -2.153545
3         -0.767366
4         -0.375842
5

dtype: float64

```

Supongamos que necesitamos el valor de relleno para variar por grupo. Una forma de hacerlo es agrupando los datos y utilizando apply con una función que llame a fillna en cada fragmento de datos. Aquí tenemos algunos datos de ejemplo sobre estados de EE. UU. divididos en regiones oriental y occidental:

```

In [106]: states = ["Ohio", "New York", "Vermont", "Florida",
.....:      "Oregon", "Nevada", "California", "Idaho"]

In [107]: group_key = ["East", "East", "East", "East",
.....:      "West", "West", "West", "West"]

In [108]: data      = pd.Series(np.random.standard_normal(8),
index=states)

In [109]: data
Out[109]:


Ohio          0.329939
New York     0.981994
Vermont      1.105913

```

```
Florida          -1.613716
Oregon           1.561587
Nevada            0.406510
California        0.359244
Idaho             -0.614436
dtype: float64
```

Establezcamos algunos valores en los datos que van a faltar:

```
In [110]: data[["Vermont", "Nevada", "Idaho"]] = np.nan
```

```
In [111]: data
```

```
Out[111]:
```

```
Ohio              0.329939
New York          0.981994
Vermont            NaN
Florida           -1.613716
Oregon            1.561587
Nevada             NaN
California         0.359244
Idaho              NaN
```

```
dtype: float64
```

```
In [112]: data.groupby(group_key).size()
```

```
Out[112]:
```

```
East               4
West               4
```

```
dtype: int64
```

```
In [113]: data.groupby(group_key).count()
```

```
Out[113]:
```

```
East               3
West               2
```

```
dtype: int64
```

```
In [114]: data.groupby(group_key).mean()
```

```
Out[114]:
```

```
-0.100594
```

```
East  
West          0.960416
```

```
dtype: float64
```

Podemos rellenar los valores NA utilizando las medias de grupos, del siguiente modo:

```
In [115]: def fill_mean(group):  
.....:     return group.fillna(group.mean())
```

```
In [116]: data.groupby(group_key).apply(fill_mean)  
Out[116]:
```

Ohio	0.329939
New York	0.981994
Vermont	-0.100594
Florida	-1.613716
Oregon	1.561587
Nevada	0.960416
California	0.359244
Idaho	0.960416

```
dtype: float64
```

En otro caso, podríamos tener en el código valores de relleno predefinidos que variaran por grupo. Como los grupos tienen un atributo name fijado internamente, podemos usarlo:

```
In [117]: fill_values = {"East": 0.5, "West": -1}
```

```
In [118]: def fill_func(group):
```

```
.....:     return group.fillna(fill_values[group.name])
```

```
In [119]: data.groupby(group_key).apply(fill_func)  
Out[119]:
```

Ohio	0.329939
New York	0.981994
Vermont	0.500000
Florida	-1.613716
Oregon	1.561587
Nevada	-1.000000
California	0.359244

```
Idaho           -1.000000
dtype: float64
```

Ejemplo: Muestreo aleatorio y permutación

Supongamos que queremos dibujar una muestra aleatoria (con o sin reemplazo) a partir de un conjunto de datos grande para la simulación de Montecarlo u otra aplicación. Hay distintas formas de realizar los «dibujos»; aquí emplearemos el método de ejemplo para el objeto Series.

A modo de demostración, esta es una forma de construir una baraja de cartas inglesa:

```
suits = ["H", "S", "C", "D"] # corazones, picas, tréboles, diamantes
card_val = (list(range(1, 11)) + [10] * 3) * 4

base_names = ["A"] + list(range(2, 11)) + ["J", "K", "Q"]
cards = []
for suit in suits:

    cards.extend(str(num) + suit for num in base_names)

deck = pd.Series(card_val, index=cards)
```

Ahora tenemos una serie de longitud 52 cuyo índice contiene nombres de cartas, y los valores son los utilizados en el blackjack y otros juegos (para simplificar las cosas, dejaré que el as “A” sea el 1):

```
In [121]: deck.head(13)
Out[121]:
```

AH	1
2H	2
3H	3
4H	4
5H	5
6H	6
7H	7
8H	8
9H	9
10H	10
JH	10
KH	10
QH	10

```
dtype: int64
```

A continuación, basándonos en lo que dije antes, el código para dibujar una mano de cinco cartas de la baraja podría ser algo así:

```
In [122]: def draw(deck, n=5):
```

```
.....:             return deck.sample(n)
```

```
In [123]: draw(deck)
```

```
Out[123]:
```

4D	4
QH	10
8S	8
7D	7
9C	9

```
dtype: int64
```

Supongamos que queríamos dos cartas aleatorias de cada palo. Como el palo es el último carácter de cada nombre de carta, podemos agrupar basándonos en esto y usar apply:

```
In [124]: def get_suit(card):
```

```
.....:             # la última carta es suit
.....:             return card[-1]
```

```
In [125]: deck.groupby(get_suit).apply(draw, n=2)
```

```
Out[125]:
```

C	6C	6
	KC	10
D	7D	7
	3D	3
H	7H	7
	9H	9
S	2S	2
	QS	10

```
dtype: int64
```

Como alternativa podemos pasar group_keys=False para quitar el índice de palo exterior, dejando solamente las cartas seleccionadas:

```
In [126]: deck.groupby(get_suit, group_keys=False).apply(draw, n=2)
Out[126]:
```

AC	1
3C	3
5D	5
4D	4
10H	10
7H	7
QS	10
7S	7

dtype: int64

Ejemplo: media ponderada de grupo y correlación

Bajo el paradigma *split-apply-combine* de groupby, son posibles las operaciones entre columnas de un dataframe o dos series, como, por ejemplo, una media ponderada de grupos. Como ejemplo, tomemos este conjunto de datos que contiene claves de grupos, valores y algunos pesos:

```
In [127]: df = pd.DataFrame({"category": ["a", "a", "a", "a",
.....:         "b", "b", "b", "b"],
.....:         "data": np.random.standard_normal(8),
.....:         "weights": np.random.uniform(size=8)})
```

```
In [128]: df
Out[128]:
```

	category	data	weights
0	a	-1.691656	0.955905
1	a	0.511622	0.012745
2	a	-0.401675	0.137009
3	a	0.968578	0.763037
4	b	-1.818215	0.492472
5	b	0.279963	0.832908
6	b	-0.200819	0.658331
7	b	-0.217221	0.612009

La media ponderada según category sería entonces:

```
In [129]: grouped = df.groupby("category")
In [130]: def get_wavg(group):
....:     return np.average(group["data"],
weights=group["weights"])
In [131]: grouped.apply(get_wavg)
Out[131]:
category
a              -0.495807
b              -0.357273
dtype: float64
```

Como otro ejemplo, veamos un conjunto de datos financiero obtenido de Yahoo! Finance que contiene precios de cierre de algunas acciones y el índice S&P 500 (el símbolo SPX):

```
In [132]: close_px      = pd.read_csv("examples/stock_px.csv",
parse_dates=True,
....: index_col=0)

In [133]: close_px.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2214 entries, 2003-01-02 to 2011-10-14
Data columns (total 4 columns):
 #   Column           Non-Null Count   Dtype  
--- 
 0   AAPL            2214 non-null    float64
 1   MSFT            2214 non-null    float64
 2   XOM             2214 non-null    float64
 3   SPX             2214 non-null    float64
dtypes: float64(4)
memory usage: 86.5 KB

In [134]: close_px.tail(4)
Out[134]:
```

	AAPL	MSFT	XOM	SPX
--	------	------	-----	-----

2011-10-11	400.29	27.00	76.27	1195.54
2011-10-12	402.19	26.96	77.16	1207.25
2011-10-13	408.43	27.18	76.37	1203.66
2011-10-14	422.00	27.27	78.11	1224.58

El método `info()` del objeto DataFrame es aquí un método práctico para obtener una visión general del contenido de un drataframe.

Una tarea interesante podría ser calcular un dataframe que consistiera en las correlaciones anuales de los rendimientos diarios (calculadas desde cambios de porcentaje) con SPX. Como una forma de hacer esto, primero creamos una función que calcule la correlación por pares de cada columna con la columna “SPX”:

```
In [135]: def spx_corr(group):
    .....
    return group.corrwith(group[“SPX”])
```

A continuación, calculamos el cambio de porcentaje sobre `close_px` utilizando `pct_change`:

```
In [136]: rets = close_px.pct_change().dropna()
```

Por último, agrupamos estos cambios de porcentaje por año, que podemos extraer desde cada etiqueta de fila con una función de una sola línea que devuelve el atributo `year` de cada etiqueta `datetime`:

```
In [137]: def get_year(x):
    .....
    return x.year
```

```
In [138]: by_year = rets.groupby(get_year)
```

```
In [139]: by_year.apply(spx_corr)
Out[139]:
```

	AAPL	MSFT	XOM	SPX
2003	0.541124	0.745174	0.661265	1.0
2004	0.374283	0.588531	0.557742	1.0
2005	0.467540	0.562374	0.631010	1.0
2006	0.428267	0.406126	0.518514	1.0
2007	0.508118	0.658770	0.786264	1.0
2008	0.681434	0.804626	0.828303	1.0
2009	0.707103	0.654902	0.797921	1.0
2010	0.710105	0.730118	0.839057	1.0

```
2011          0.691931          0.800996          0.859975      1.0
```

También podríamos calcular correlaciones entre columnas. Aquí calculamos la correlación anual entre Apple y Microsoft:

```
In [140]: def corr_aapl_msft(group):
....:     return group[“AAPL”].corr(group[“MSFT”])

In [141]: by_year.apply(corr_aapl_msft)
Out[141]:
2003          0.480868
2004          0.259024
2005          0.300093
2006          0.161735
2007          0.417738
2008          0.611901
2009          0.432738
2010          0.571946
2011          0.58198 7
dtype: float64
```

Ejemplo: Regresión lineal por grupos

Siguiendo el mismo tema que el del ejemplo anterior, podemos usar groupby para realizar análisis estadísticos por grupos más complejos, siempre que la función devuelva un objeto pandas o un valor escalar. Por ejemplo, puedo definir la siguiente función regress (utilizando la librería de econometría statsmodels), que ejecuta una regresión de mínimos cuadrados ordinarios (MCO) en cada fragmento de datos:

```
import statsmodels.api as sm
def regress(data, yvar=None, xvars=None):
    Y = data[yvar]
    X = data[xvars]
    X[“intercept”] = 1.
    result = sm.OLS(Y, X).fit()
    return result.params
```

Se puede instalar statsmodels con conda si no lo está ya:

```
conda install statsmodels
```

Ahora, para realizar una regresión lineal anual de AAPL sobre rendimientos SPX, ejecutamos el siguiente código:

```
In [143]: by_year.apply(regress, yvar="AAPL", xvars=["SPX"])
Out[143]:
```

	SPX	intercept
2003	1.195406	0.000710
2004	1.363463	0.004201
2005	1.766415	0.003246
2006	1.645496	0.000080
2007	1.198761	0.003438
2008	0.968016	-0.001110
2009	0.879103	0.002954
2010	1.052608	0.001261
2011	0.806605	0.001514

10.4 Transformaciones de grupos y funciones GroupBy «simplificadas»

En la sección 10.3 «El método apply: un *split-apply-combine* general», vimos el método `apply` en operaciones agrupadas para realizar transformaciones. Hay otro método interno denominado `transform`, similar a `apply`, pero impone más limitaciones en el tipo de función que permite utilizar:

- Puede producir un valor escalar que se transmite a la forma del grupo.
- Puede producir un objeto con la misma forma que el grupo de entrada.
- No debe mutar su entrada.

Consideremos un sencillo ejemplo que nos sirve para ilustrar esto:

```
In [144]: df = pd.DataFrame({'key': ['a', 'b', 'c'] * 4,
.....: 'value': np.arange(12.)})
```

```
In [145]: df
Out[145]:
```

	key	value
0	a	0.0
1	b	1.0
2	c	2.0
3	a	3.0
4	b	4.0

```
5          c      5.0
6          a      6.0
7          b      7.0
8          c      8.0
9          a      9.0
10         b     10.0
11         c     11.0
```

Estas son las medias de grupos por clave:

```
In [146]: g = df.groupby('key')['value']
```

```
In [147]: g.mean()
```

```
Out[147]:
```

```
key
```

```
a      4.5
b      5.5
c      6.5
```

```
Name: value, dtype: float64
```

Supongamos que lo que realmente queríamos era producir una serie con la misma forma que `df['value']` pero con los valores sustituidos por la media agrupada según 'key'. Podemos pasarle a `transform` una función que calcule la media de un solo grupo:

```
In [148]: def get_mean(group):
.....: return group.mean()
```

```
In [149]: g.transform(get_mean)
```

```
Out[149]:
```

```
0      4.5
1      5.5
2      6.5
3      4.5
4      5.5
5      6.5
6      4.5
7      5.5
8      6.5
9      4.5
10     5.5
11     6.5
```

```
Name: value, dtype: float64
```

Para funciones de agregación internas, podemos pasar un alias de cadena de texto como con el método `agg` de `GroupBy`:

```
In [150]: g.transform('mean')
Out[150]:
```

0	4.5
1	5.5
2	6.5
3	4.5
4	5.5
5	6.5
6	4.5
7	5.5
8	6.5
9	4.5
10	5.5
11	6.5

```
Name: value, dtype: float64
```

Al igual que `apply`, `transform` sirve perfectamente con funciones que devuelven objetos Series, pero el resultado debe tener el mismo tamaño que la entrada. Por ejemplo, podemos multiplicar cada grupo por 2 utilizando una función auxiliar:

```
In [151]: def times_two(group):
    ....:
    return group * 2

In [152]: g.transform(times_two)
Out[152]:
```

0	0.0
1	2.0
2	4.0
3	6.0
4	8.0
5	10.0
6	12.0
7	14.0
8	16.0
9	18.0

```
10          20.0  
11          22.0
```

Name: value, dtype: float64

Como ejemplo más complicado, podemos calcular los rangos en orden descendente para cada grupo:

```
In [153]: def get_ranks(group):  
.....:         return group.rank(ascending=False)
```

```
In [154]: g.transform(get_ranks)  
Out[154]:
```

```
0          4.0  
1          4.0  
2          4.0  
3          3.0  
4          3.0  
5          3.0  
6          2.0  
7          2.0  
8          2.0  
9          1.0  
10         1.0  
11         1.0
```

Name: value, dtype: float64

Veamos una función de transformación de grupo creada a partir de agregaciones sencillas:

```
In [155]: def normalize(x):  
.....:         return (x-x.mean()) / x.std()
```

Podemos obtener resultados equivalentes en este caso utilizando `transform` o `apply`:

```
In [156]: g.transform(normalize)  
Out[156]:
```

```
0          -1.161895
1          -1.161895
2          -0.387298
3          -0.387298
4          -0.387298
5          -0.387298
6           0.387298
7           0.387298
8           0.387298
9           1.161895
10          1.161895
11          1.161895
```

Name: value, dtype: float64

```
In [157]: g.apply(normalize)
Out[157]:
```

```
-1.161895
0          -1.161895
1          -1.161895
2          -0.387298
3          -0.387298
4          -0.387298
5          -0.387298
6           0.387298
7           0.387298
8           0.387298
9           1.161895
10          1.161895
11          1.161895
```

Name: value, dtype: float64

Las funciones de agregación internas como ‘mean’ o ‘sum’ suelen ser mucho más rápidas que una función apply general. También ofrecen un «camino rápido» cuando se utilizan con transform, lo que nos permite realizar lo que se denomina operación de grupo simplificada:

```
In [158]: g.transform('mean')
Out[158]:
```

0	4.5
1	5.5
2	6.5

```
3          4.5
4          5.5
5          6.5
6          4.5
7          5.5
8          6.5
9          4.5
10         5.5
11         6.5
```

Name: value, dtype: float64

```
In [159]: normalized = (df['value']-g.transform('mean')) / g.transform('std')
```

```
In [160]: normalized
Out[160]:
```

```
0          -1.161895
1          -1.161895
2          -1.161895
3          -0.387298
4          -0.387298
5          -0.387298
6           0.387298
7           0.387298
8           0.387298
9           1.161895
10          1.161895
11          1.161895
```

Name: value, dtype: float64

Aquí estamos haciendo cálculos entre los resultados de múltiples operaciones GroupBy en lugar de escribir una función y pasarla a `groupby(...).apply`. A esto es a lo que nos referimos con «simplificado».

Aunque una operación de grupos simplificada pueda implicar varias agregaciones de grupos, el beneficio global que suponen las operaciones vectorizadas suele pesar más para su uso.

10.5 Tablas dinámicas y tabulación cruzada

Una tabla dinámica es una herramienta para resumir datos que se suele encontrar en programas de hoja de cálculo y software de análisis de datos similar. Agrega una tabla de datos según una o varias claves, disponiendo los datos en un rectángulo con algunas de las claves de grupos a lo largo de las filas y otras a lo largo de las columnas. Las tablas dinámicas en Python son posibles con pandas utilizando el método `groupby` descrito en este capítulo, combinado con operaciones de remodelado que emplean indexación jerárquica. El objeto `DataFrame` tiene también un método `pivot_table`, igual que existe una función de mismo nivel `pandas.pivot_table`. Además de ofrecer una interfaz adecuada para `groupby`, `pivot_table` puede añadir totales parciales (con el parámetro `margins`).

Volviendo al conjunto de datos de las propinas, supongamos que queremos calcular una tabla de medias de grupos (el tipo de agregación predeterminada `pivot_table`) organizada por `day` y `smoker` según las filas:

```
In [161]: tips.head()
Out[161]:
```

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.059447
1	10.34	1.66	No	Sun	Dinner	3	0.160542
2	21.01	3.50	No	Sun	Dinner	3	0.166587
3	23.68	3.31	No	Sun	Dinner	2	0.139780
4	24.59	3.61	No	Sun	Dinner	4	0.146808

```
In [162]: tips.pivot_table(index=["day", "smoker"])
Out[162]:
```

day	smoker	size	tip	tip_pct	total_bill
		No	2.250000	2.812500	0.151650
Fri	Yes	2.066667	2.714000	0.174783	16.813333
Sat	No	2.555556	3.102889	0.158048	19.661778
	Yes	2.476190	2.875476	0.147906	21.276667
Sun	No	2.929825	3.167895	0.160113	20.506667
	Yes	2.578947	3.516842	0.187250	24.120000
Thur	No	2.488889	2.673778	0.160298	17.113111
	Yes	2.352941	3.030000	0.163863	19.190588

Esto se podría haber hecho directamente con `groupby`, utilizando `tips.groupby(["day", "smoker"]).mean()`. Ahora supongamos que queremos tomar

la media solo de tip_pct y size, y agrupar por time de forma adicional. Añadiré smoker a las columnas de la tabla y time y day a las filas:

```
In [163]: tips.pivot_table(index=["time", "day"], columns="smoker",
.....:           values=["tip_pct", "size"])

Out[163]:
```

smoker	time	day	size		tip_pct	
			No	Yes	No	Yes
Dinner	Fri	2.000000	2.222222	0.139622	0.165347	
		2.555556	2.476190	0.158048	0.147906	
		2.929825	2.578947	0.160113	0.187250	
		2.000000	NaN	0.159744	NaN	
Lunch	Fri	3.000000	1.833333	0.187735	0.188937	
	Thur	2.500000	2.352941	0.160311	0.163863	

Podríamos ampliar esta tabla para que incluya totales parciales pasando margins=True. Con esto logramos añadir etiquetas de fila y columna All, siendo los valores correspondientes las estadísticas de grupos para todos los datos con una sola capa:

```
In [164]: tips.pivot_table(index=["time", "day"], columns="smoker",
.....:           values=["tip_pct", "size"], margins=True)

Out[164]:
```

smoker	time	day	size		tip_pct		
			No	Yes	All	No	Yes
Dinner	Fri	2.000000	2.222222	2.166667	0.139622	0.165347	0.158916
		2.555556	2.476190	2.517241	0.158048	0.147906	0.153152
		2.929825	2.578947	2.842105	0.160113	0.187250	0.166897
		2.000000	NaN	2.000000	0.159744	NaN	0.159744
Lunch	Fri	3.000000	1.833333	2.000000	0.187735	0.188937	0.188765
	Thur	2.500000	2.352941	2.459016	0.160311	0.163863	0.161301
All		2.668874	2.408602	2.569672	0.159328	0.163196	0.160803

Aquí, los valores All son promedios que no tienen en cuenta que se sea fumador o no (las columnas All) o alguno de los dos niveles de agrupamiento de las filas (la fila All).

Para utilizar una función de agregación distinta a mean, se la pasamos al argumento de palabra clave aggfunc. Por ejemplo, “count” o len nos dará una tabulación cruzada (recuento o frecuencia) de tamaños de grupos (aunque “count” excluirá los valores nulos obtenidos del recuento que haya en los grupos de datos, mientras que len no lo hará):

```
In [165]: tips.pivot_table(index=["time", "smoker"], columns="day",
....:             values="tip_pct", aggfunc=len, margins=True)

Out[165]:
```

day		Fri	Sat	Sun	Thur	All
time	smoker					
Dinner	No	3.0	45.0	57.0	1.0	106
	Yes	9.0	42.0	19.0	NaN	70
Lunch	No	1.0	NaN	NaN	44.0	45
	Yes	6.0	NaN	NaN	17.0	23
All		19.0	87.0	76.0	62.0	244

Si algunas combinaciones están vacías (o son NA), quizá convenga pasar un fill_value:

```
In [166]: tips.pivot_table(index=["time", "size", "smoker"],
columns="day",
....:             values="tip_pct", fill_value=0)

Out[166]:
```

time	size	smoker	day	Fri	Sat	Sun	Thur
Dinner	1	No	0.000000	0.137931	0.000000	0.000000	0.000000
		Yes	0.000000	0.325733	0.000000	0.000000	0.000000
	2	No	0.139622	0.162705	0.168859	0.159744	
		Yes	0.171297	0.148668	0.207893	0.000000	
	3	No	0.000000	0.154661	0.152663	0.000000	
		Yes	0.000000	0.000000	0.000000	0.204952	
...		

4	No	0.000000	0.000000	0.000000	0.138919
	Yes	0.000000	0.000000	0.000000	0.155410
5	No	0.000000	0.000000	0.000000	0.121389
6	No	0.000000	0.000000	0.000000	0.173706

[21 rows x 4 columns]

La tabla 10.2 ofrece un resumen de las opciones de `pivot_table`.

Tabla 10.2. Opciones de pivot_table.

Argumento	Descripción
values	Nombre o nombres de columna para agregar; de forma predeterminada, agrega todas las columnas numéricas.
index	Nombres de columna u otras claves de grupos para agrupar según las filas de la tabla dinámica resultante.
columns	Nombres de columna u otras claves de grupos para agrupar según las columnas de la tabla dinámica resultante.
aggfunc	Función o lista de funciones de agregación ("mean" por defecto); puede ser cualquier función válida en un contexto groupby.
fill_value	Reemplaza valores ausentes en la tabla de resultados.
dropna	Si es True, no incluye columnas cuyas entradas son todas NA.
margins	Añade subtotales de fila/columna y total general (False por defecto).
margins_name	Nombre que se utiliza para las etiquetas de fila/columna cuando se pasa margins=True; el valor predeterminado es "All".
observed	Con claves de grupos categóricas, si es True solo muestra los valores de categoría observados en las claves en lugar de todas las categorías.

Tabulaciones cruzadas

Una tabulación cruzada es un caso especial de tabla dinámica que calcula frecuencias de grupos. Aquí tenemos un ejemplo:

In [167]: `from io import StringIO`

In [168]: `data = """Sample Nationality Handedness`

.....:	1	USA	Right-handed
.....:	2	Japan	Left-handed
.....:	3	USA	Right-handed

```

.....:      4      Japan      Right-handed
.....:      5      Japan      Left-handed
.....:      6      Japan      Right-handed
.....:      7      USA       Right-handed
.....:      8      USA       Left-handed
.....:      9      Japan      Right-handed
.....:     10      USA      Right-handed"""
.....:

```

In [169]: `data = pd.read_table(StringIO(data), sep="\s+")`

In [170]: `data`
Out[170]:

Sample Nationality Handedness

0	1	USA	Right-handed
1	2	Japan	Left-handed
2	3	USA	Right-handed
3	4	Japan	Right-handed
4	5	Japan	Left-handed
5	6	Japan	Right-handed
6	7	USA	Right-handed
7	8	USA	Left-handed
8	9	Japan	Right-handed
9	10	USA	Right-handed

Como parte de posibles análisis de encuestas, nos podría interesar resumir estos datos por nacionalidad y mano dominante. Se podría utilizar `pivot_table` para hacer esto, pero la función `pandas.crosstab` sería más práctica:

In [171]: `pd.crosstab(data["Nationality"], data["Handedness"], margins=True)`
Out[171]:

Nationality \ Handedness	Left-handed	Right-handed	All
Japan	2	3	5
USA	1	4	5
All	3	7	10

Los primeros dos argumentos de `crosstab` pueden ser un array o una serie o una lista de arrays. Igual que en los datos de las propinas:

```
In [172]: pd.crosstab([tips["time"], tips["day"]], tips["smoker"], margins=True)
Out[172]:
```

smoker	day	No	Yes	All
Dinner	Fri	3	9	12
	Sat	45	42	87
	Sun	57	19	76
	Thur	1	0	1
Lunch	Fri	1	6	7
	Thur	44	17	61
All		151	93	244

10.6 Conclusión

Dominar las herramientas de agrupamiento de datos de pandas facilita la limpieza y modelado de los datos o el trabajo de análisis estadístico. En el capítulo 13 veremos más ejemplos de casos de uso de groupby con datos reales.

En el siguiente capítulo centraremos nuestra atención en los datos de series temporales.

Capítulo 11

Series temporales

Los datos de series temporales son una forma importante de datos estructurados en muchos campos distintos, como las finanzas, la economía, la ecología, la neurociencia y la física. Cualquier cosa que se registre repetidamente en muchos puntos en el tiempo forma una serie temporal. Muchas series temporales tienen una frecuencia fija, es decir, los puntos de datos ocurren a intervalos regulares según alguna regla, como, por ejemplo, cada 15 segundos, cada 5 minutos o una vez al mes. Las series temporales también pueden ser irregulares, sin unidad de tiempo fija ni desfase entre unidades. El modo en que se marcan los datos de series temporales y se hace referencia a ellos depende de la aplicación, así que podríamos tener cualesquiera de los siguientes:

- Marcas temporales: Instantes específicos en el tiempo.
- Periodos fijos: Como, por ejemplo, el mes entero de enero de 2017, o todo el año 2020.
- Intervalos de tiempo: Indicados por una marca temporal inicial y final. Los períodos pueden considerarse como casos especiales de intervalos.
- Tiempo transcurrido o tiempo del experimento: Cada marca temporal es una medida de tiempo relativa a un determinado momento inicial (por ejemplo, el diámetro de una galleta que se hornea cada segundo desde que es introducida en el horno), empezando por 0.

En este capítulo me interesan principalmente las series temporales de las tres primeras categorías, aunque es posible aplicar muchas de las técnicas a las series temporales de tiempo del experimento, en las cuales el índice puede ser un entero o un número en punto flotante, que indica el tiempo transcurrido desde el inicio del experimento. El tipo más sencillo de serie temporal está indexado por marca temporal.



pandas soporta también índices basados en el tipo `timedelta`, que puede ser una forma útil de representar tiempos del experimento o tiempos transcurridos. No exploraremos índices `timedelta` en

este libro, pero si el lector lo desea, puede investigar más en la documentación de pandas (<https://pandas.pydata.org>).

En pandas encontramos muchas herramientas y algoritmos de series temporales. Se puede trabajar de una manera eficaz con grandes series temporales, y segmentar, agregar y remuestrear series temporales irregulares y de frecuencia fija. Algunas de estas herramientas son útiles para aplicaciones financieras y económicas, pero, sin duda, se pueden utilizar perfectamente para analizar datos de registros de servidor.

Como con el resto de los capítulos, empezamos importando NumPy y pandas:

```
In [12]: import numpy as np
```

```
In [13]: import pandas as pd
```

11.1 Tipos de datos de fecha y hora y herramientas asociadas

La librería estándar de Python incluye tipos de datos para fechas y horas, además de funcionalidad relacionada con el calendario. Los módulos `datetime`, `time` y `calendar` son los lugares principales para empezar. El tipo `datetime.datetime`, o simplemente `datetime`, se utiliza mucho:

```
In [14]: from datetime import datetime
```

```
In [15]: now = datetime.now()
```

```
In [16]: now
```

```
Out[16]: datetime.datetime(2022, 8, 12, 14, 9, 11, 337033)
```

```
In [17]: now.year, now.month, now.day
```

```
Out[17]: (2022, 8, 12)
```

El tipo `datetime` almacena la fecha y hora hasta el microsegundo, mientras que `datetime.timedelta`, o simplemente `timedelta`, representa la diferencia de tiempo entre dos objetos `datetime`:

```
In [18]: delta = datetime(2011, 1, 7)-datetime(2008, 6, 24, 8, 15)
```

```
In [19]: delta
```

```
Out[19]: datetime.timedelta(days=926, seconds=56700)
```

```
In [20]: delta.days  
Out[20]: 926
```

```
In [21]: delta.seconds  
Out[21]: 56700
```

Se puede sumar (o restar) un `timedelta` o varios a un objeto `datetime` para obtener un nuevo objeto desplazado en el tiempo:

```
In [22]: from datetime import timedelta  
  
In [23]: start = datetime(2011, 1, 7)  
  
In [24]: start + timedelta(12)  
Out[24]: datetime.datetime(2011, 1, 19, 0, 0)  
  
In [25]: start - 2 * timedelta(12)  
Out[25]: datetime.datetime(2010, 12, 14, 0, 0)
```

La tabla 11.1 resume los tipos de datos del módulo `datetime`. Aunque este capítulo se centra principalmente en los tipos de datos de pandas y en la manipulación de series temporales de máximo nivel, es posible encontrar los tipos basados en `datetime` en muchos otros lugares de Python.

Tabla 11.1. Tipos del módulo `datetime`.

Tipo	Descripción
<code>date</code>	Almacena la fecha del calendario (año, mes, día) utilizando el calendario gregoriano.
<code>time</code>	Almacena la hora del día como horas, minutos, segundos y microsegundos.
<code>Datetime</code>	Almacena la fecha y la hora.
<code>timedelta</code>	La diferencia entre dos valores <code>datetime</code> (como días, segundos y microsegundos).
<code>tzinfo</code>	Tipo básico para almacenar información de zona horaria.

Conversión entre cadena de texto y `datetime`

Se pueden formatear objetos `datetime` y objetos `Timestamp` de pandas (que explicaré más adelante) como cadenas de texto utilizando `str` o el método

`strftime`, pasando una especificación de formato:

```
In [26]: stamp = datetime(2011, 1, 3)
```

```
In [27]: str(stamp)
Out[27]: '2011-01-03 00:00:00'
```

```
In [28]: stamp.strftime("%Y-%m-%d")
Out[28]: '2011-01-03'
```

La tabla 11.2 ofrece una completa lista de códigos de formato.

Tabla 11.2. Especificación de formato `datetime` (compatible con ISO C89).

Tipo	Descripción
%Y	Año de cuatro dígitos.
%y	Año de dos dígitos.
%m	Mes de dos dígitos [01, 12].
%d	Día de dos dígitos [01, 31].
%H	Hora (formato de 24 horas) [00, 23].
%I	Hora (formato de 12 horas) [01, 12].
%M	Minuto de dos dígitos [00, 59].
%S	Segundo [00, 61] (60 segundos, 61 para segundos bisiestos).
%f	Microsegundo como un entero, llenado con ceros (de 000000 a 999999).
%j	Día del año como entero llenado con ceros (de 001 a 336).
%w	Día de la semana como entero [0 (domingo), 6].
%u	Día de la semana como entero empezando por 1, donde 1 es lunes.
%U	Número de semana del año [00, 53]; el domingo se considera el primer día de la semana, y los días que preceden al primer domingo del año son la «semana 0».
%W	Número de semana del año [00, 53]; el lunes se considera el primer día de la semana, y los días que preceden al primer lunes del año son la «semana 0».
%z	Desfase de zona horaria UTC como +HHMM o -HHMM; queda vacío si no se es consciente de la zona horaria.
%Z	Nombre de la zona horaria como cadena de texto, o cadena de texto vacío si no hay zona horaria.

Tipo	Descripción
%F	Abreviatura de %Y-%m-%d (por ejemplo, 2012-04-18).
%D	Abreviatura de %m/%d/%Y (por ejemplo, 04/18/2012).

Se pueden utilizar muchos de los mismos códigos de formato para convertir cadenas de texto en fechas utilizando `datetime.strptime` (pero algunos códigos, como `%F`, no se pueden usar):

```
In [29]: value = "2011-01-03"

In [30]: datetime.strptime(value, "%Y-%m-%d")
Out[30]: datetime.datetime(2011, 1, 3, 0, 0)

In [31]: datestrs = ["7/6/2011", "8/6/2011"]

In [32]: [datetime.strptime(x, "%m/%d/%Y") for x in datestrs]
Out[32]:

[datetime.datetime(2011, 7, 6, 0, 0),
 datetime.datetime(2011, 8, 6, 0, 0)]
```

El método `datetime.strptime` es una forma de analizar una fecha con un formato conocido.

pandas está normalmente orientado al trabajo con arrays de fechas, ya se utilicen como el índice de un eje o como una columna de un dataframe. El método `pandas.to_datetime` analiza muchos tipos distintos de representaciones de fecha. Los formatos de fecha estándares, como ISO 8601, se pueden analizar rápidamente:

```
In [33]: datestrs = ["2011-07-06 12:00:00", "2011-08-06
00:00:00"]

In [34]: pd.to_datetime(datestrs)
Out[34]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06
00:00:00'], dtype='datetime64[ns]', freq=None)
```

Tambien gestiona valores que se deberían considerar ausentes (`None`, cadena de texto vacía, etc.):

```
In [35]: idx = pd.to_datetime(datestrs + [None])
```

```
In [36]: idx
Out[36]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06
00:00:00', 'NaT'], dtype='datetime64[ns]', freq=None)

In [37]: idx[2]
Out[37]: NaT

In [38]: pd.isna(idx)
Out[38]: array([False, False, True])
```

NaT (Not a Time) es el valor nulo de pandas para datos de marca temporal.



`dateutil.parser` es una herramienta útil, aunque imperfecta. Es decir, reconocerá algunas cadenas de texto como fechas que podríamos preferir que no lo hiciera; por ejemplo, “42” se analizará como el año 2042 con la fecha de calendario actual.

Los objetos `datetime` tienen también una serie de opciones de formato de configuración regional para sistemas de otros países o idiomas. Por ejemplo, los nombres de meses abreviados serán diferentes en los sistemas alemán o francés comparados con los sistemas anglosajones. La tabla 11.3 muestra un listado.

Tabla 11.3. Formato de fechas según la configuración regional.

Tipo	Descripción
%a	Nombre del día de la semana abreviado.
%A	Nombre del día de la semana completo.
%b	Nombre del mes abreviado.
%B	Nombre del mes completo.
%c	Fecha y hora completas (por ejemplo, en inglés 'Tue 01 May 2012 04:20:57 PM').
%p	Equivalente local de AM o PM.
%x	Fecha con formato adecuado a la localidad (por ejemplo, en los Estados Unidos, May 1, 2012 produce '05/01/2012').
%X	Hora con formato adecuado a la localidad (por ejemplo, '04:24:12 PM').

11.2 Fundamentos de las series temporales

Un tipo básico de objeto serie temporal en pandas es una serie indexada por marcas temporales, que a menudo se representa fuera de pandas como una cadena de texto Python o como un objeto datetime:

```
In [39]: dates = [datetime(2011, 1, 2), datetime(2011, 1, 5),
....: datetime(2011, 1, 7), datetime(2011, 1, 8),
....: datetime(2011, 1, 10), datetime(2011, 1, 12)]
In [40]: ts = pd.Series(np.random.standard_normal(6),
index=dates)
In [41]: ts
Out[41]:
```

2011-01-02	-0.204708
2011-01-05	0.478943
2011-01-07	-0.519439
2011-01-08	-0.555730
2011-01-10	1.965781
2011-01-12	1.393406

dtype: float64

Técnicamente, estos objetos datetime se han colocado en un DatetimeIndex:

```
In [42]: ts.index
Out[42]:
```

```
DatetimeIndex(['2011-01-02', '2011-01-05', '2011-01-07', '2011-01-08',
'2011-01-10', '2011-01-12'],
dtype='datetime64[ns]', freq=None)
```

Igual que cualquier otro objeto Series, las operaciones aritméticas entre distintas series temporales indexadas de distinta manera se alinean según las fechas:

```
In [43]: ts + ts[::2]
Out[43]:
```

```
2011-01-02           -0.409415
2011-01-05            NaN
2011-01-07          -1.038877
2011-01-08            NaN
2011-01-10          3.931561
2011-01-12            NaN

dtype: float64
```

Recordemos que `ts[::2]` selecciona cada segundo elemento de `ts`.

En pandas las marcas temporales se almacenan utilizando el tipo de datos `datetime64` de NumPy a la resolución de nanosegundo:

```
In [44]: ts.index.dtype
Out[44]: dtype('<M8[ns]')
```

Los valores escalares de un `DatetimeIndex` son objetos `Timestamp` de pandas:

```
In [45]: stamp = ts.index[0]
In [46]: stamp
Out[46]: Timestamp('2011-01-02 00:00:00')
```

Un objeto `pandas.Timestamp` se puede sustituir en la mayoría de los sitios en los que se utilizaría un objeto `datetime`. Pero a la inversa es imposible, porque `pandas.Timestamp` puede almacenar datos con precisión de nanosegundo, mientras que `datetime` solo almacena hasta microsegundos. Además, `pandas.Timestamp` puede almacenar información de frecuencias (si es que la hay) y comprende cómo realizar conversiones de zona horaria y otros tipos de manipulaciones. En la sección 11.4 «Manipulación de zonas horarias» doy más información sobre esto.

Indexación, selección y creación de subconjuntos

Las series temporales se comportan como cualquier otro objeto `Series` cuando se están indexando y seleccionando datos basados en la etiqueta:

```
In [47]: stamp = ts.index[2]
```

```
In [48]: ts[stamp]  
Out[48]: -0.5194387150567381
```

Por comodidad se puede pasar también una cadena que puede interpretarse como una fecha:

```
In [49]: ts["2011-01-10"]  
Out[49]: 1.9657805725027142
```

Para series temporales más largas, se puede pasar un año o solo un año y un mes para seleccionar fácilmente segmentos de datos (hablaré con más detalle sobre pandas.date_range en el apartado «Generación de rangos de fecha», más adelante en el capítulo):

```
In [50]: longer_ts = pd.Series(np.random.standard_normal(1000),  
..... index=pd.date_range("2000-01-01", periods=1000))
```

```
In [51]: longer_ts  
Out[51]:
```

2000-01-01	0.092908
2000-01-02	0.281746
2000-01-03	0.769023
2000-01-04	1.246435
2000-01-05	1.007189
...	...
2002-09-22	0.930944
2002-09-23	-0.811676
2002-09-24	-1.830156
2002-09-25	-0.138730
2002-09-26	0.334088

Freq: D, Length: 1000, dtype: float64

```
In [52]: longer_ts["2001"]  
Out[52]:
```

2001-01-01	1.599534
2001-01-02	0.474071
2001-01-03	0.151326
2001-01-04	-0.542173
2001-01-05	-0.475496

```
2001-12-27          ...          0.057874  
2001-12-28          -0.433739  
2001-12-29          0.092698  
2001-12-30          -1.397820  
2001-12-31          1.457823
```

Freq: D, Length: 365, dtype: float64

Aquí, la cadena de texto “2001” se interpreta como un año, así que se selecciona ese periodo de tiempo. Esto también funciona si se especifica el mes:

```
In [53]: longer_ts["2001-05"]  
Out[53]:
```

```
2001-05-01          -0.622547  
2001-05-02          0.936289  
2001-05-03          0.750018  
2001-05-04          -0.056715  
2001-05-05          2.300675  
...  
2001-05-27          0.235477  
2001-05-28          0.111835  
2001-05-29          -1.251504  
2001-05-30          -2.949343  
2001-05-31          0.634634
```

Freq: D, Length: 31, dtype: float64

Segmentar con objetos datetime también funciona:

```
In [54]: ts[datetime(2011, 1, 7):]  
Out[54]:
```

```
2011-01-07          -0.519439  
2011-01-08          -0.555730  
2011-01-10          1.965781  
2011-01-12          1.393406
```

dtype: float64

```
In [55]: ts[datetime(2011, 1, 7):datetime(2011, 1, 10)]  
Out[55]:
```

2011-01-07	-0.519439
2011-01-08	-0.555730
2011-01-10	1.965781

dtype: float64

Como la mayoría de los datos de series temporales está ordenada cronológicamente, se puede segmentar con marcas temporales no contenidas en una serie temporal para realizar la consulta de un rango:

```
In [56]: ts  
Out[56]:
```

2011-01-02	-0.204708
2011-01-05	0.478943
2011-01-07	-0.519439
2011-01-08	-0.555730
2011-01-10	1.965781
2011-01-12	1.393406

dtype: float64

```
In [57]: ts["2011-01-06":"2011-01-11"]  
Out[57]:
```

2011-01-07	-0.519439
2011-01-08	-0.555730
2011-01-10	1.965781

dtype: float64

Como antes, se puede pasar una fecha de cadena de texto, un datetime o una marca temporal. Recordemos que segmentar de este modo produce vistas según la serie temporal de origen, igual que cuando se fragmentan arrays NumPy. Esto significa que no se copian datos, y que las modificaciones de la segmentación se verán reflejadas en los datos originales.

Hay un método de instancia equivalente, `truncate`, que divide un objeto Series entre dos fechas:

```
In [58]: ts.truncate(after="2011-01-09")
Out[58]:
```

2011-01-02	-0.204708
2011-01-05	0.478943
2011-01-07	-0.519439
2011-01-08	-0.555730

dtype: float64

Todo esto se aplica igualmente a objetos DataFrame, indexando según sus filas:

```
In [59]: dates = pd.date_range("2000-01-01", periods=100,
freq="W-WED")

In [60]: long_df = pd.DataFrame(np.random.standard_normal((100,
4)),
```

....: index=dates,
....: columns=[“Colorado”, “Texas”,
....: “New York”, “Ohio”])

```
In [61]: long_df.loc[“2001-05”]
Out[61]:
```

	Colorado	Texas	New York	Ohio
2001-05-02	-0.006045	0.490094	-0.277186	-0.707213
2001-05-09	-0.560107	2.735527	0.927335	1.513906
2001-05-16	0.538600	1.273768	0.667876	-0.969206
2001-05-23	1.676091	-0.817649	0.050188	1.951312
2001-05-30	3.260383	0.963301	1.201206	-1.852001

Series temporales con índices duplicados

En algunas aplicaciones se pueden producir varias observaciones de datos que entran dentro de una determinada marca temporal. Aquí tenemos un

ejemplo:

```
In [62]: dates = pd.DatetimeIndex(["2000-01-01", "2000-01-02",
"2000-01-02",
....: "2000-01-02", "2000-01-03"])

In [63]: dup_ts = pd.Series(np.arange(5), index=dates)

In [64]: dup_ts
Out[64]:
```

2000-01-01	0
2000-01-02	1
2000-01-02	2
2000-01-02	3
2000-01-03	4

dtype: int64

Podemos decir que el índice no es único comprobando su propiedad `is_unique`:

```
In [65]: dup_ts.index.is_unique
Out[65]: False
```

Indexar dentro de esta serie temporal producirá ahora valores escalares o bien segmentos, dependiendo de si una marca temporal está duplicada:

```
In [66]: dup_ts["2000-01-03"] # no duplicada
Out[66]: 4
```

```
In [67]: dup_ts["2000-01-02"] # duplicada
Out[67]:
```

2000-01-02	1
2000-01-02	2
2000-01-02	3

dtype: int64

Supongamos que queremos agregar los datos que tienen marcas temporales no únicas. Una forma de hacerlo es utilizando `groupby` y pasando `level=0` (el único nivel existente):

```
In [68]: grouped = dup_ts.groupby(level=0)
```

```
In [69]: grouped.mean()  
Out[69]:
```

2000-01-01	0.0
2000-01-02	2.0
2000-01-03	4.0

```
dtype: float64
```

```
In [70]: grouped.count()  
Out[70]:
```

2000-01-01	1
2000-01-02	3
2000-01-03	1

```
dtype: int64
```

11.3 Rangos de fechas, frecuencias y desplazamiento

Las series temporales genéricas en pandas se suponen irregulares, es decir, no tienen una frecuencia fija. Para muchas aplicaciones esto es suficiente, pero a veces es deseable trabajar en relación con una frecuencia fija, como diaria, mensual o cada 15 minutos, incluso si ello significa introducir valores ausentes en una serie temporal. Afortunadamente, pandas incluye un juego completo de frecuencias y herramientas estándares de series temporales para remuestrear (que veremos con detalle en la sección 11.6 «Remuestreo y conversión de frecuencias»), inferir frecuencias y generar rangos de fechas de frecuencia fija. Por ejemplo, es posible convertir la serie temporal de muestra en una frecuencia diaria fija llamando a `resample`:

```
In [71]: ts  
Out[71]:
```

2011-01-02	-0.204708
2011-01-05	0.478943
2011-01-07	-0.519439
2011-01-08	-0.555730

```
2011-01-10          1.965781
2011-01-12          1.393406
```

dtype: float64

```
In [72]: resampler = ts.resample("D")
```

```
In [73]: resampler
```

```
Out[73]: <pandas.core.resample.DatetimeIndexResampler object at
0x7febd896bc40>
```

La cadena de texto “D” se interpreta como frecuencia diaria.

La conversión entre frecuencias o remuestreo es un tema de la importancia suficiente como para tener más adelante su propia sección (11.6 «Remuestreo y conversión de frecuencias»). En ella mostraré cómo utilizar las frecuencias de base y sus múltiplos.

Generación de rangos de fechas

Aunque ya lo he utilizado anteriormente sin dar una explicación, `pandas.date_range` es responsable de generar un `DatetimeIndex` con una longitud indicada según una determinada frecuencia:

```
In [74]: index = pd.date_range("2012-04-01", "2012-06-01")
```

```
In [75]: index
```

```
Out[75]:
```

```
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',
                 '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
                 '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
                 '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
                 '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20',
                 '2012-04-21', '2012-04-22', '2012-04-23', '2012-04-24',
                 '2012-04-25', '2012-04-26', '2012-04-27', '2012-04-28',
                 '2012-04-29', '2012-04-30', '2012-05-01', '2012-05-02',
                 '2012-05-03', '2012-05-04', '2012-05-05', '2012-05-06',
                 '2012-05-07', '2012-05-08', '2012-05-09', '2012-05-10',
                 '2012-05-11', '2012-05-12', '2012-05-13', '2012-05-14',
                 '2012-05-15', '2012-05-16', '2012-05-17', '2012-05-18',
                 '2012-05-19', '2012-05-20', '2012-05-21', '2012-05-22',
                 '2012-05-23', '2012-05-24', '2012-05-25', '2012-05-26',
                 '2012-05-27', '2012-05-28', '2012-05-29', '2012-05-30'],
                dtype='datetime64[ns]', freq='D')
```

```
          30',      01',      02',
'2012-05-03', '2012-05-  '2012-05-  '2012-05-
              04',      05',      06',
'2012-05-07', '2012-05-  '2012-05-  '2012-05-
              08',      09',      10',
'2012-05-11', '2012-05-  '2012-05-  '2012-05-
              12',      13',      14',
'2012-05-15', '2012-05-  '2012-05-  '2012-05-
              16',      17',      18',
'2012-05-19', '2012-05-  '2012-05-  '2012-05-
              20',      21',      22',
'2012-05-23', '2012-05-  '2012-05-  '2012-05-
              24',      25',      26',
'2012-05-27', '2012-05-  '2012-05-  '2012-05-
              28',      29',      30',
'2012-05-31', '2012-06-  '2012-06-  '2012-06-
              01'],
dtype='datetime64[ns]', freq='D')
```

De forma predeterminada, `pandas.date_range` genera marcas temporales diarias. Si se pasa solamente una fecha de inicio o de fin, se deben pasar el número de periodos que se deseen generar:

```
In [76]: pd.date_range(start="2012-04-01", periods=20)
Out[76]:
```

```
DatetimeIndex ([ '2012-04-
                  01',      '2012-04-
                  02',      '2012-04-
                  03',      '2012-04-
                  04',
                  '2012-04-05', '2012-04-
                  06',      '2012-04-
                  07',      '2012-04-
                  08',
                  '2012-04-09', '2012-04-
                  10',      '2012-04-
                  11',      '2012-04-
                  12',
                  '2012-04-13', '2012-04-
                  14',      '2012-04-
                  15',      '2012-04-
                  16',
                  '2012-04-17', '2012-04-
                  18',      '2012-04-
                  19',      '2012-04-
                  20'],
dtype='datetime64[ns]', freq='D')
```

```
In [77]: pd.date_range(end="2012-06-01", periods=20)
Out[77]:
```

```
DatetimeIndex ([ '2012-05-
                  13',      '2012-05-
                  14',      '2012-05-
                  15',      '2012-05-
                  16',
```

```

        '2012-05-17',   '2012-05-    '2012-05-    '2012-05-
                      18',       19',       20',
        '2012-05-21',   '2012-05-    '2012-05-    '2012-05-
                      22',       23',       24',
        '2012-05-25',   '2012-05-    '2012-05-    '2012-05-
                      26',       27',       28',
        '2012-05-29',   '2012-05-    '2012-05-    '2012-06-
                      30',       31',       01'],
dtype='datetime64[ns]', freq='D')

```

Las fechas de inicio y fin definen estrictos límites para el índice de fecha generado. Por ejemplo, si quisieramos un índice de fecha que contuviera el último día laborable de cada mes, pasaríamos la frecuencia “BM” (fin de mes laborable; la tabla 11.4 ofrece un listado completo de frecuencias), y solo se incluirían las fechas que entraran dentro de ese intervalo de fechas:

```
In [78]: pd.date_range("2000-01-01", "2000-12-01", freq="BM")
Out[78]:
```

```

DatetimeIndex(['2000-01-31', '2000-02-29', '2000-03-31', '2000-04-
                '2000-05-31', '2000-06-30', '2000-07-31', '2000-08-
                '2000-09-29', '2000-10-31', '2000-11-30'],
dtype='datetime64[ns]', freq='BM')

```

Tabla 11.4. Frecuencias básicas de series temporales (listado no exhaustivo).

Alias	Tipo de desfase	Descripción
D	Day	Cada día natural
B	BusinessDay	Cada día laborable
H	Hour	Cada hora
T or min	Minute	Una vez por minuto
S	Second	Una vez por segundo
L or ms	Milli	Milisegundo (1/1.000 de segundo)

Alias	Tipo de desfase	Descripción
U	Micro	Microsegundo (1/1.000.000 de segundo)
M	MonthEnd	Último día natural del mes
BM	BusinessMonthEnd	Último día laborable del mes
MS	MonthBegin	Primer día natural del mes
BMS	BusinessMonthBegin	Primer día laborable del mes
W- MON, W- TUE, ...	Week	Semanalmente en un determinado día de la semana (MON para lunes, TUE para martes, WED para miércoles, THU para jueves, FRI para viernes, SAT para sábado o SUN para domingo).
WOM- 1MOM, WOM- 2MON, ...	WeekOfMonth	Genera fechas semanalmente en la primera, segunda, tercera, o cuarta semana del mes (por ejemplo, WOM-3FRI para el tercer viernes de cada mes).
Q- JAN, Q- FEB, ...	QuarterEnd	Fechas trimestrales ancladas en el último día natural de cada mes, por año que termina en el mes indicado (JAN para enero, FEB para febrero, MAR para marzo, APR para abril, MAY para mayo, JUN para julio, JUL para julio, AUG para agosto, SEP para septiembre, OCT para octubre, NOV para noviembre o DEC para diciembre).
QS- JAN, QS- FEB, ...	QuarterBegin	Fechas trimestrales ancladas en el primer día laborable de cada mes, por año que termina en el mes indicado.
BQS- JAN, BQS- FEB, ...	BusinessQuarterBegin	Fechas trimestrales ancladas en el primer día laborable de cada mes, por año que termina en el mes indicado.
A- JAN, A- FEB, ...	YearEnd	Fechas anuales ancladas en el último día natural del mes dado (JAN para enero, FEB para febrero, MAR para marzo, APR para abril, MAY para mayo, JUN para julio, JUL para julio, AUG para agosto, SEP para septiembre, OCT para octubre, NOV para noviembre o DEC para diciembre).

Alias	Tipo de desfase	Descripción
BA-JAN, BA-FEB, ...	BusinessYearEnd	Fechas anuales ancladas en el último día laborable del mes dado.
AS-JAN, AS-FEB, ...	YearBegin	Fechas anuales ancladas en el primer día del mes dado.
BAS-JAN, BAS-FEB, ...	BusinessYearBegin	Fechas anuales ancladas en el primer día laborable del mes dado.

pandas.date_range conserva por defecto la hora (si la hay) del inicio o fin de la marca temporal:

```
In [79]: pd.date_range("2012-05-02 12:56:31", periods=5)
Out[79]:
```

```
DatetimeIndex(['2012-05-02 12:56:31', '2012-05-03 12:56:31',
'2012-05-04 12:56:31', '2012-05-05 12:56:31',
'2012-05-06 12:56:31'],
dtype='datetime64[ns]', freq='D')
```

En ocasiones tendremos fechas de inicio o fin con información de hora, pero nos interesará generar un conjunto de marcas temporales normalizadas a medianoche como convenio. Para ello existe una opción normalize:

```
In [80]: pd.date_range("2012-05-02 12:56:31", periods=5,
normalize=True)
Out[80]:
```

```
DatetimeIndex(['2012-05-02', '2012-05-03', '2012-05-04', '2012-05-05',
'2012-05-06'],
```

```
dtype='datetime64[ns]', freq='D')
```

Frecuencias y desfases de fechas

Las frecuencias en pandas están formadas por una frecuencia base y un multiplicador. Las frecuencias base se suelen indicar con un alias de cadena de texto, por ejemplo “M” para una frecuencia mensual o “H” para una frecuencia horaria. Para cada frecuencia base tenemos un objeto al que denominamos desfase de fecha. Por ejemplo, la frecuencia por horas se puede representar con la clase Hour:

```
In [81]: from pandas.tseries.offsets import Hour, Minute
```

```
In [82]: hour = Hour()
```

```
In [83]: hour
```

```
Out[83]: <Hour>
```

Se puede definir un múltiplo de un desfase pasando un entero:

```
In [84]: four_hours = Hour(4)
```

```
In [85]: four_hours
```

```
Out[85]: <4 * Hours>
```

En la mayoría de las aplicaciones, nunca tendría que ser necesario crear explícitamente uno de estos objetos, sino que bastaría con emplear un alias de cadena de texto como “H” o “4H”. Poner un entero antes de la frecuencia base crea un múltiplo:

```
In [86]: pd.date_range("2000-01-01", "2000-01-03 23:59",
freq="4H")
Out[86]:
```

```
DatetimeIndex ([ '2000-01-01 00:00:00', '2000-01-01 04:00:00',
'2000-01-01 08:00:00', '2000-01-01 12:00:00',
'2000-01-01 16:00:00', '2000-01-01 20:00:00',
'2000-01-02 00:00:00', '2000-01-02 04:00:00',
'2000-01-02 08:00:00', '2000-01-02 12:00:00',
'2000-01-02 16:00:00', '2000-01-02 20:00:00',
'2000-01-03 00:00:00', '2000-01-03 04:00:00',
'2000-01-03 08:00:00', '2000-01-03 12:00:00',
```

```
'2000-01-03 16:00:00', '2000-01-03 20:00:00'],
dtype='datetime64[ns]', freq='4H')
```

Muchos desfases pueden combinarse sumándolos:

```
In [87]: Hour(2) + Minute(30)
Out[87]: <150 * Minutes>
```

De forma similar, se pueden pasar cadenas de texto de frecuencia, como, por ejemplo, “1h30min”, que se analizarán efectivamente en la misma expresión:

```
In [88]: pd.date_range("2000-01-01", periods=10, freq="1h30min")
Out[88]:
```

```
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 01:30:00',
                 '2000-01-01 03:00:00', '2000-01-01 04:30:00',
                 '2000-01-01 06:00:00', '2000-01-01 07:30:00',
                 '2000-01-01 09:00:00', '2000-01-01 10:30:00',
                 '2000-01-01 12:00:00', '2000-01-01 13:30:00'],
                dtype='datetime64[ns]', freq='90T')
```

Algunas frecuencias describen puntos en el tiempo que no están espaciados por igual. Por ejemplo, “M” (fin de mes natural) y “BM” (último día laborable del mes) dependen del número de días del mes y, en el último caso, de si el mes termina en fin de semana o no. Estas frecuencias se denominan desfases anclados.

La tabla 11.4 muestra un listado con los códigos de frecuencia y las clases de desfase de fechas disponibles en pandas.



Los usuarios pueden definir sus propias clases de frecuencia personalizadas para disponer de lógica de fechas no incluida en pandas, pero los detalles completos de este proceso quedan fuera del alcance de este libro.

Fechas de la semana del mes

Una clase de frecuencia útil es «semana del mes», que empieza por WOM (Week Of Month: semana del mes). Permite obtener fechas como, por ejemplo, el tercer viernes de cada mes:

```
In [89]: monthly_dates = pd.date_range("2012-01-01", "2012-09-01", freq="WOM-3FRI")  
  
In [90]: list(monthly_dates)  
Out[90]:  
[Timestamp('2012-01-20 00:00:00', freq='WOM-3FRI'),  
 Timestamp('2012-02-17 00:00:00', freq='WOM-3FRI'),  
 Timestamp('2012-03-16 00:00:00', freq='WOM-3FRI'),  
 Timestamp('2012-04-20 00:00:00', freq='WOM-3FRI'),  
 Timestamp('2012-05-18 00:00:00', freq='WOM-3FRI'),  
 Timestamp('2012-06-15 00:00:00', freq='WOM-3FRI'),  
 Timestamp('2012-07-20 00:00:00', freq='WOM-3FRI'),  
 Timestamp('2012-08-17 00:00:00', freq='WOM-3FRI')]
```

Desplazamiento de los datos (adelantar y retrasar)

El término «desplazamiento» se refiere a mover datos hacia atrás y hacia adelante en el tiempo. Tanto los objetos Series como DataFrame tienen un método shift para realizar desplazamientos sencillos hacia delante o hacia atrás, dejando el índice sin modificar:

```
In [91]: ts = pd.Series(np.random.standard_normal(4),  
 ....: index=pd.date_range("2000-01-01", periods=4, freq="M"))  
  
In [92]: ts  
Out[92]:
```

2000-01-31	-0.066748
2000-02-29	0.838639
2000-03-31	-0.117388
2000-04-30	-0.517795

Freq: M, dtype: float64

```
In [93]: ts.shift(2)  
Out[93]:
```

2000-01-31	NaN
2000-02-29	NaN

```
2000-03-31 -0.066748  
2000-04-30 0.838639
```

Freq: M, dtype: float64

```
In [94]: ts.shift(-2)  
Out[94]:
```

```
2000-01-31 -0.117388  
2000-02-29 -0.517795  
2000-03-31 NaN  
2000-04-30 NaN
```

Freq: M, dtype: float64

Cuando realizamos desplazamientos como estos, se introducen datos ausentes o bien al principio o al final de la serie temporal.

Un uso habitual de shift es para calcular cambios de porcentaje consecutivos en una o varias series temporales como columnas de un dataframe. Esto se expresa así:

```
ts / ts.shift(1) - 1
```

Como los desplazamientos no conscientes de la zona horaria dejan el índice sin modificar, algunos datos se eliminan. Así, si la frecuencia es conocida, se le puede pasar a shift para adelantar las marcas temporales en lugar de pasar simplemente los datos:

```
In [95]: ts.shift(2, freq="M")  
Out[95]:
```

```
2000-03-31 -0.066748  
2000-04-30 0.838639  
2000-05-31 -0.117388  
2000-06-30 -0.517795
```

Freq: M, dtype: float64

También se pueden pasar otras frecuencias, proporcionando así una cierta flexibilidad en el modo de adelantar y retrasar los datos:

```
In [96]: ts.shift(3, freq="D")
Out[96]:
2000-02-03           -0.066748
2000-03-03            0.838639
2000-04-03           -0.117388
2000-05-03           -0.517795
dtype: float64

In [97]: ts.shift(1, freq="90T")
Out[97]:
2000-01-31 01:30:00           -0.066748
2000-02-29 01:30:00            0.838639
2000-03-31 01:30:00           -0.117388
2000-04-30 01:30:00           -0.517795
dtype: float64
```

Aquí la τ significa minutos. Tengamos en cuenta que el parámetro freq indica en este caso el desfase que se debe aplicar a las marcas temporales, pero no cambia la frecuencia subyacente de los datos, si es que existe.

Desplazamiento de fechas con desfases

Los desfases de fecha de pandas se pueden utilizar también con objetos datetime o Timestamp:

```
In [98]: from pandas.tseries.offsets import Day, MonthEnd
In [99]: now = datetime(2011, 11, 17)
In [100]: now + 3 * Day()
Out[100]: Timestamp('2011-11-20 00:00:00')
```

Si se añade un desfase anclado como MonthEnd, el primer incremento adelantará la fecha a la siguiente según la regla de frecuencia marcada:

```
In [101]: now + MonthEnd()
Out[101]: Timestamp('2011-11-30 00:00:00')
In [102]: now + MonthEnd(2)
```

```
Out[102]: Timestamp('2011-12-31 00:00:00')
```

Los desfases anclados pueden adelantar o atrasar fechas de forma explícita simplemente empleando sus métodos `rollforward` y `rollback`, respectivamente:

```
In [103]: offset = MonthEnd()
```

```
In [104]: offset.rollforward(now)
```

```
Out[104]: Timestamp('2011-11-30 00:00:00')
```

```
In [105]: offset.rollback(now)
```

```
Out[105]: Timestamp('2011-10-31 00:00:00')
```

Los defases de fecha se pueden emplear de forma creativa usando estos métodos con `groupby`:

```
In [106]: ts = pd.Series(np.random.standard_normal(20),
```

```
.....:     index=pd.date_range("2000-01-15",           periods=20,  
                                freq="4D"))
```

```
In [107]: ts
```

```
Out[107]:
```

2000-01-15	-0.116696
2000-01-19	2.389645
2000-01-23	-0.932454
2000-01-27	-0.229331
2000-01-31	-1.140330
2000-02-04	0.439920
2000-02-08	-0.823758
2000-02-12	-0.520930
2000-02-16	0.350282
2000-02-20	0.204395
2000-02-24	0.133445
2000-02-28	0.327905
2000-03-03	0.072153
2000-03-07	0.131678
2000-03-11	-1.297459
2000-03-15	0.997747
2000-03-19	0.870955
2000-03-23	-0.991253

```
2000-03-27          0.151699
2000-03-31          1.266151
```

Freq: 4D, dtype: float64

```
In [108]: ts.groupby(MonthEnd()).rollforward().mean()
Out[108]:
```

```
2000-01-31          -0.005833
2000-02-29          0.015894
2000-03-31          0.150209
```

dtype: float64

Por supuesto, con `resample` esto es mucho más sencillo y rápido (lo trataremos en profundidad en la sección 11.6 «Remuestreo y conversión de frecuencias»):

```
In [109]: ts.resample("M").mean()
Out[109]:
```

```
2000-01-31          -0.005833
2000-02-29          0.015894
2000-03-31          0.150209
```

Freq: M, dtype: float64

11.4 Manipulación de zonas horarias

El trabajo con zonas horarias puede ser una de las partes menos agradables de la manipulación de series temporales. Por esta razón muchos usuarios de series temporales eligen trabajar con ellas en tiempo UTC, o tiempo coordinado universal (*Coordinated Universal Time*), el estándar internacional geográficamente independiente. Las zonas horarias se expresan como desfases partiendo del estándar UTC; por ejemplo, Nueva York va cuatro horas atrasada con respecto al tiempo UTC durante el horario de verano y cinco horas el resto del año.

En Python, la información sobre zonas horarias procede de la librería externa `pytz`, que expone la base de datos Olson, una compilación de información sobre

las zonas horarias mundiales. Esto es de especial importancia en caso de datos históricos, dado que las fechas de transición de verano e invierno (e incluso los desfases UTC) han sido modificados varias veces dependiendo de las leyes regionales. En los Estados Unidos, los horarios de verano e invierno han cambiado muchas veces desde 1900.

Para más información sobre la librería pytz conviene echar un vistazo a su documentación correspondiente. En lo que a este libro se refiere, pandas incluye la funcionalidad de pytz, de modo que es posible ignorar su API fuera de los nombres de las zonas horarias. Como pandas depende mucho de pytz, no hace falta instalarlo por separado (se puede instalar con pip o conda). Los nombres de las zonas horarias se pueden encontrar interactivamente y en los documentos:

```
In [110]: import pytz
```

```
In [111]: pytz.common_timezones[-5:]  
Out[111]: ['US/Eastern', 'US/Hawaii', 'US/Mountain',  
          'US/Pacific', 'UTC']
```

Para obtener un objeto de zona horaria de pytz usamos pytz.timezone:

```
In [112]: tz = pytz.timezone("America/New_York")
```

```
In [113]: tz  
Out[113]: <DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

Los métodos de pandas aceptarán los nombres de las zonas horarias o estos objetos.

Localización y conversión de zonas horarias

De forma predeterminada, las series temporales de pandas no son conscientes de las zonas horarias. Por ejemplo, veamos la siguiente serie temporal:

```
In [114]: dates = pd.date_range("2012-03-09 09:30", periods=6)
```

```
In [115]: ts = pd.Series(np.random.standard_normal(len(dates)),  
index=dates)
```

```
In [116]: ts  
Out[116]:
```

2012-03-09	09:30:00	
2012-03-10	09:30:00	0.050718
2012-03-11	09:30:00	0.639869
2012-03-12	09:30:00	0.597594
2012-03-13	09:30:00	-0.797246
2012-03-14	09:30:00	0.472879

Freq: D, dtype: float64

El campo tz del índice es None:

In [117]: print(ts.index.tz)

None

Se pueden generar rangos de fechas con una zona horaria fijada:

In [118]: pd.date_range("2012-03-09 09:30", periods=10, tz="UTC")
Out[118]:

```
DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:00',
                 '2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
                 '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',
                 '2012-03-15 09:30:00+00:00', '2012-03-16 09:30:00+00:00',
                 '2012-03-17 09:30:00+00:00', '2012-03-18 09:30:00+00:00'],
                dtype='datetime64[ns, UTC]', freq='D')
```

La conversión de no consciente a localizado (reinterpretada como que se ha observado en una determinada zona horaria) se gestiona con el método tz_localize:

In [119]: ts
Out[119]:

2012-03-09	09:30:00	-0.202469
2012-03-10	09:30:00	0.050718

2012-03-11	09:30:00	0.639869
2012-03-12	09:30:00	0.597594
2012-03-13	09:30:00	-0.797246
2012-03-14	09:30:00	0.472879

Freq: D, dtype: float64

In [120]: ts_utc = ts.tz_localize("UTC")

In [121]: ts_utc

Out[121]:

2012-03-09	09:30:00+00:00	-0.202469
2012-03-10	09:30:00+00:00	0.050718
2012-03-11	09:30:00+00:00	0.639869
2012-03-12	09:30:00+00:00	0.597594
2012-03-13	09:30:00+00:00	-0.797246
2012-03-14	09:30:00+00:00	0.472879

Freq: D, dtype: float64

In [122]: ts_utc.index

Out[122]:

```
DatetimeIndex(['2012-03-09 09:30:00', '2012-03-10 09:30:00',
                '2012-03-11 09:30:00', '2012-03-12 09:30:00',
                '2012-03-13 09:30:00', '2012-03-14 09:30:00'],
               dtype='datetime64[ns, UTC]', freq='D')
```

Una vez que una serie se ha localizado en una determinada zona horaria, se puede convertir a otra con tz_convert:

In [123]: ts_utc.tz_convert("America/New_York")

Out[123]:

2012-03-09	04:30:00-05:00	-0.202469
2012-03-10	04:30:00-05:00	0.050718
2012-03-11	05:30:00-04:00	0.639869
2012-03-12	05:30:00-04:00	0.597594

```
2012-03-13      05:30:00-04:00      -0.797246
2012-03-14      05:30:00-04:00      0.472879
```

Freq: D, dtype: float64

En el caso de la serie temporal anterior, que está en la transición del horario de verano de la zona horaria America/New_York, podríamos localizarla en la hora del este de EE. UU y convertirla, por ejemplo, a tiempo UTC o a la hora de Berlín:

```
In [124]: ts_eastern = ts.tz_localize("America/New_York")
```

```
In [125]: ts_eastern.tz_convert("UTC")
Out[125]:
```

```
2012-03-09      14:30:00+00:00      -0.202469
2012-03-10      14:30:00+00:00      0.050718
2012-03-11      13:30:00+00:00      0.639869
2012-03-12      13:30:00+00:00      0.597594
2012-03-13      13:30:00+00:00      -0.797246
2012-03-14      13:30:00+00:00      0.472879
```

dtype: float64

```
In [126]: ts_eastern.tz_convert("Europe/Berlin")
Out[126]:
```

```
2012-03-09      15:30:00+01:00      -0.202469
2012-03-10      15:30:00+01:00      0.050718
2012-03-11      14:30:00+01:00      0.639869
2012-03-12      14:30:00+01:00      0.597594
2012-03-13      14:30:00+01:00      -0.797246
2012-03-14      14:30:00+01:00      0.472879
```

dtype: float64

`tz_localize` y `tz_convert` son también métodos de instancia en `DatetimeIndex`:

```
In [127]: ts.index.tz_localize("Asia/Shanghai")
Out[127]:
```

```
DatetimeIndex(['2012-03-09 09:30:00+08:00', '2012-03-10 09:30:00+08:00',  
                '2012-03-11 09:30:00+08:00', '2012-03-12 09:30:00+08:00',  
                '2012-03-13 09:30:00+08:00', '2012-03-14 09:30:00+08:00'],  
               dtype='datetime64[ns, Asia/Shanghai]', freq=None)
```



La localización de marcas temporales no conscientes comprueba también las horas ambiguas o no existentes en torno a las transiciones del horario de verano/invierno.

Operaciones con objetos de marca temporal conscientes de la zona horaria

Más o menos como con las series temporales y los rangos de fechas, los objetos `Timestamp` individuales pueden transformarse de no conscientes a conscientes de la zona horaria y ser convertidos de una zona horaria a otra:

```
In [128]: stamp = pd.Timestamp("2011-03-12 04:00")  
  
In [129]: stamp_utc = stamp.tz_localize("utc")  
  
In [130]: stamp_utc.tz_convert("America/New_York")  
Out[130]: Timestamp('2011-03-11 23:00:00-0500',  
tz='America/New_York')
```

También se puede pasar una zona horaria al crear el objeto `Timestamp`:

```
In [131]: stamp_moscow = pd.Timestamp("2011-03-12 04:00",  
tz="Europe/Moscow")  
  
In [132]: stamp_moscow  
Out[132]: Timestamp('2011-03-12 04:00:00+0300',  
tz='Europe/Moscow')
```

Los objetos `Timestamp` conscientes de la zona horaria almacenan internamente un valor de marca temporal UTC como nanosegundos desde el epoch de Unix (1 de enero de 1970), de modo que cambiar la zona horaria no altera el valor UTC interno:

```
In [133]: stamp_utc.value  
Out[133]: 12999024000000000000  
  
In [134]: stamp_utc.tz_convert("America/New_York").value
```

```
Out[134]: 12999024000000000000
```

Cuando se realizan operaciones aritméticas temporales utilizando los objetos DateOffset de pandas, pandas respeta las transiciones de horario de verano/invierno todo lo posible. En este caso vamos a construir marcas temporales que ocurren justo antes que las transiciones (hacia delante y hacia atrás). Primero, 30 minutos antes de cambiar a la hora de verano:

```
In [135]: stamp = pd.Timestamp("2012-03-11 01:30", tz="US/Eastern")
```

```
In [136]: stamp  
Out[136]: Timestamp('2012-03-11 01:30:00-0500', tz='US/Eastern')
```

```
In [137]: stamp + Hour()  
Out[137]: Timestamp('2012-03-11 03:30:00-0400', tz='US/Eastern')
```

Después, 90 minutos antes de cambiar al horario de invierno:

```
In [138]: stamp = pd.Timestamp("2012-11-04 00:30", tz="US/Eastern")
```

```
In [139]: stamp  
Out[139]: Timestamp('2012-11-04 00:30:00-0400', tz='US/Eastern')
```

```
In [140]: stamp + 2 * Hour()  
Out[140]: Timestamp('2012-11-04 01:30:00-0500', tz='US/Eastern')
```

Operaciones entre distintas zonas horarias

Si dos series temporales con distintas zonas horarias se combinan, el resultado será UTC. Como las marcas temporales se almacenan internamente en UTC, es una operación directa y no requiere conversión:

```
In [141]: dates = pd.date_range("2012-03-07 09:30", periods=10, freq="B")
```

```
In [142]: ts = pd.Series(np.random.standard_normal(len(dates)), index=dates)
```

```
In [143]: ts  
Out[143]:
```

2012-03-07	09:30:00	0.522356
------------	----------	----------

2012-03-08	09:30:00	-0.546348
2012-03-09	09:30:00	-0.733537
2012-03-12	09:30:00	1.302736
2012-03-13	09:30:00	0.022199
2012-03-14	09:30:00	0.364287
2012-03-15	09:30:00	-0.922839
2012-03-16	09:30:00	0.312656
2012-03-19	09:30:00	-1.128497
2012-03-20	09:30:00	-0.333488

Freq: B, dtype: float64

```
In [144]: ts1 = ts[:7].tz_localize("Europe/London")
```

```
In [145]: ts2 = ts1[2:].tz_convert("Europe/Moscow")
```

```
In [146]: result = ts1 + ts2
```

```
In [147]: result.index
```

```
Out[147]:
```

```
DatetimeIndex(['2012-03-07 09:30:00+00:00', '2012-03-08 09:30:00+00:00',
                 '2012-03-09 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
                 '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',
                 '2012-03-15 09:30:00+00:00'],
                dtype='datetime64[ns, UTC]', freq=None)
```

Las operaciones entre datos no conscientes y conscientes de la zona horaria no se soportan y producirán un error.

11.5 Periodos y aritmética de períodos

Los períodos representan lapsos de tiempo, como días, meses, trimestres o años. La clase pandas.Period representa este tipo de datos, y requiere una cadena de texto o un entero y una frecuencia soportada de la tabla 11.4:

```
In [148]: p = pd.Period("2011", freq="A-DEC")
```

```
In [149]: p  
Out[149]: Period('2011', 'A-DEC')
```

En este caso, el objeto Period representa el lapso de tiempo completo desde el 1 de enero de 2011 hasta el 31 de diciembre de 2011, inclusive. Resulta cómodo que sumar y restar enteros a los periodos tenga como efecto desplazar su frecuencia:

```
In [150]: p + 5  
Out[150]: Period('2016', 'A-DEC')
```

```
In [151]: p - 2  
Out[151]: Period('2009', 'A-DEC')
```

Si dos periodos tienen la misma frecuencia, su diferencia es el número de unidades entre ellos como desfase de fecha:

```
In [152]: pd.Period("2014", freq="A-DEC") - p  
Out[152]: <3 * YearEnds: month=12>
```

Se pueden construir rangos regulares de periodos con la función `period_range`:

```
In [153]: periods = pd.period_range("2000-01-01", "2000-06-30",  
freq="M")
```

```
In [154]: periods  
Out[154]: PeriodIndex(['2000-01', '2000-02', '2000-03', '2000-  
04', '2000-05', '2000-06'], dtype='period[M']')
```

La clase `PeriodIndex` almacena una secuencia de periodos y puede servir como índice de eje en cualquier estructura de datos de pandas:

```
In [155]: pd.Series(np.random.standard_normal(6), index=periods)  
Out[155]:
```

2000-01	-0.514551
2000-02	-0.559782
2000-03	-0.783408
2000-04	-1.797685
2000-05	-0.172670
2000-06	0.680215

Freq: M, dtype: float64

Si tenemos un array de cadenas de texto, también podemos usar la clase `PeriodIndex`, cuyos valores son todos períodos:

```
In [156]: values = ["2001Q3", "2002Q2", "2003Q1"]  
In [157]: index = pd.PeriodIndex(values, freq="Q-DEC")  
In [158]: index  
Out[158]: PeriodIndex(['2001Q3', '2002Q2', '2003Q1'],  
dtype='period[Q-DEC']')
```

Conversión de frecuencias de períodos

Los períodos y los objetos `PeriodIndex` se pueden convertir a otra frecuencia con su método `asfreq`. Como ejemplo, supongamos que tenemos un período anual y queremos convertirlo a mensual al principio o final del año. Esto puede hacerse así:

```
In [159]: p = pd.Period("2011", freq="A-DEC")  
In [160]: p  
Out[160]: Period('2011', 'A-DEC')  
In [161]: p.asfreq("M", how="start")  
Out[161]: Period('2011-01', 'M')  
In [162]: p.asfreq("M", how="end")  
Out[162]: Period('2011-12', 'M')  
In [163]: p.asfreq("M")  
Out[163]: Period('2011-12', 'M')
```

Se puede pensar en `Period("2011", "A-DEC")` como si fuera una especie de cursor apuntando a un lapso de tiempo, subdividido en períodos mensuales. La figura 11.1 muestra una ilustración de ello. En el caso de un año fiscal que termine en un mes distinto a diciembre, los subperiodos mensuales correspondientes son distintos:

```
In [164]: p = pd.Period("2011", freq="A-JUN")  
In [165]: p  
Out[165]: Period('2011', 'A-JUN')  
In [166]: p.asfreq("M", how="start")
```

```
Out[166]: Period('2010-07', 'M')
```

```
In [167]: p.asfreq("M", how="end")
Out[167]: Period('2011-06', 'M')
```

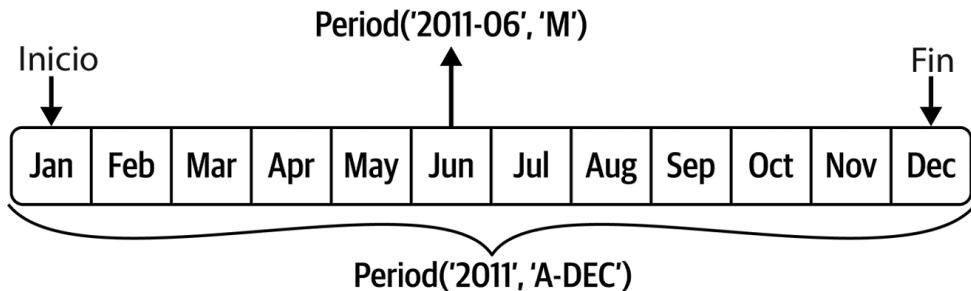


Figura 11.1. Ilustración de conversión de frecuencias de períodos.

Cuando estamos convirtiendo de frecuencia alta a baja, pandas determina el subperiodo, dependiendo de donde «pertenezca» el superperiodo. Por ejemplo, en una frecuencia A-JUN, el mes Aug-2011 es en realidad parte del periodo 2012:

```
In [168]: p = pd.Period("Aug-2011", "M")
```

```
In [169]: p.asfreq("A-JUN")
Out[169]: Period('2012', 'A-JUN')
```

Se pueden convertir objetos o series temporales PeriodIndex enteros de una forma parecida con la misma semántica:

```
In [170]: periods = pd.period_range("2006", "2009", freq="A-DEC")
```

```
In [171]: ts = pd.Series(np.random.standard_normal(len(periods)),
index=periods)
```

```
In [172]: ts
Out[172]:
```

2006	1.607578
2007	0.200381
2008	-0.834068
2009	-0.302988

Freq: A-DEC, dtype: float64

```
In [173]: ts.asfreq("M", how="start")
```

```
Out[173]:
```

2006-01	1.607578
2007-01	0.200381
2008-01	-0.834068
2009-01	-0.302988

Freq: M, dtype: float64

Aquí, los periodos anuales son reemplazados por periodos mensuales que corresponden al primer mes que caiga dentro de cada periodo anual. Pero si queremos el último día laborable de cada año, podemos usar la frecuencia “B” e indicar que queremos el final del periodo:

```
In [174]: ts.asfreq("B", how="end")  
Out[174]:
```

2006-12-29	1.607578
2007-12-31	0.200381
2008-12-31	-0.834068
2009-12-31	-0.302988

Freq: B, dtype: float64

Frecuencias de periodos trimestrales

Los datos trimestrales son estándares en contabilidad, finanzas y otros campos. Gran parte de estos datos se comunican en relación a un cierre de año fiscal, normalmente el último día natural o laborable de uno de los 12 meses del año. Es decir, el periodo 2012Q4 tiene un significado distinto dependiendo del final del año fiscal. En pandas se soportan las 12 frecuencias trimestrales posibles desde Q-JAN hasta Q-DEC:

```
In [175]: p = pd.Period("2012Q4", freq="Q-JAN")  
  
In [176]: p  
Out[176]: Period('2012Q4', 'Q-JAN')
```

En el caso de un año fiscal que termine en enero, 2012Q4 va desde noviembre de 2011 hasta enero de 2012, lo que puede verificarse convirtiendo a frecuencia diaria:

```
In [177]: p.asfreq("D", how="start")
Out[177]: Period('2011-11-01', 'D')
```

```
In [178]: p.asfreq("D", how="end")
Out[178]: Period('2012-01-31', 'D')
```

La figura 11.2 muestra una ilustración.

Así, es posible realizar cómodos cálculos aritméticos con períodos. Por ejemplo, para obtener la marca temporal a las 4 p. m. del penúltimo día hábil del trimestre, podríamos hacer lo siguiente:

```
In [179]: p4pm = (p.asfreq("B", how="end")-1).asfreq("T", how="start") + 16 * 60
```

```
In [180]: p4pm
Out[180]: Period('2012-01-30 16:00', 'T')
```

```
In [181]: p4pm.to_timestamp()
Out[181]: Timestamp('2012-01-30 16:00:00')
```



Figura 11.2. Distintos convenios de frecuencia trimestral.

El método `to_timestamp` devuelve el objeto `Timestamp` al principio del periodo de forma predeterminada.

Se pueden generar rangos trimestrales empleando `pandas.period_range`. Los cálculos son también idénticos:

```
In [182]: periods = pd.period_range("2011Q3", "2012Q4", freq="Q-JAN")
```

```
In [183]: ts = pd.Series(np.arange(len(periods)), index=periods)
```

```
In [184]: ts
```

```
Out[184]:
```

2011Q3	0
2011Q4	1
2012Q1	2
2012Q2	3
2012Q3	4
2012Q4	5

```
Freq: Q-JAN, dtype: int64
```

```
In [185]: new_periods = (periods.asfreq("B", "end") - 1).asfreq("H", "start") + 16
```

```
In [186]: ts.index = new_periods.to_timestamp()
```

```
In [187]: ts
```

```
Out[187]:
```

2010-10-28	16:00:00	0
2011-01-28	16:00:00	1
2011-04-28	16:00:00	2
2011-07-28	16:00:00	3
2011-10-28	16:00:00	4
2012-01-30	16:00:00	5

```
dtype: int64
```

Conversión de marcas temporales a periodos (y viceversa)

Los objetos Series y DataFrame indexados por series temporales pueden convertirse a periodos con el método `to_period`:

```
In [188]: dates = pd.date_range("2000-01-01", periods=3, freq="M")
```

```
In [189]: ts = pd.Series(np.random.standard_normal(3),  
index=dates)
```

```
In [190]: ts  
Out[190]:
```

2000-01-31	1.663261
2000-02-29	-0.996206
2000-03-31	1.521760

Freq: M, dtype: float64

```
In [191]: pts = ts.to_period()
```

```
In [192]: pts  
Out[192]:
```

2000-01	1.663261
2000-02	-0.996206
2000-03	1.521760

Freq: M, dtype: float64

Como los periodos hacen referencia a lapsos de tiempo que no se superponen, una marca temporal solo puede pertenecer a un único periodo para una determinada frecuencia. Aunque la frecuencia del nuevo PeriodIndex se puede deducir de las marcas temporales de forma predeterminada, es posible especificar cualquier frecuencia soportada (la mayoría de las listadas en la tabla 11.4 se soportan). Tampoco es un problema que aparezcan periodos duplicados en el resultado:

```
In [193]: dates = pd.date_range("2000-01-29", periods=6)
```

```
In [194]: ts2 = pd.Series(np.random.standard_normal(6),  
index=dates)
```

```
In [195]: ts2  
Out[195]:
```

2000-01-29	0.244175
2000-01-30	0.423331

```
2000-01-31          -0.654040
2000-02-01           2.089154
2000-02-02          -0.060220
2000-02-03          -0.167933
```

Freq: D, dtype: float64

```
In [196]: ts2.to_period("M")
Out[196]:
```

```
2000-01              0.244175
2000-01              0.423331
2000-01          -0.654040
2000-02           2.089154
2000-02          -0.060220
2000-02          -0.167933
```

Freq: M, dtype: float64

Para volver a convertir a marcas temporales, usamos el método `to_timestamp`, que devuelve un objeto `DatetimeIndex`:

```
In [197]: pts = ts2.to_period()
```

```
In [198]: pts
Out[198]:
```

```
2000-01-29          0.244175
2000-01-30           0.423331
2000-01-31          -0.654040
2000-02-01           2.089154
2000-02-02          -0.060220
2000-02-03          -0.167933
```

Freq: D, dtype: float64

```
In [199]: pts.to_timestamp(how="end")
Out[199]:
```

```
2000-01-29 23:59:59.999999999 0.244175
2000-01-30 23:59:59.999999999 0.423331
2000-01-31 23:59:59.999999999 -0.654040
```

```

2000-02-01          23:59:59.999999999         2.089154
2000-02-02          23:59:59.999999999        -0.060220
2000-02-03          23:59:59.999999999        -0.167933

```

Freq: D, dtype: float64

Creación de un objeto PeriodIndex a partir de arrays

Los conjuntos de datos de frecuencia fija se almacenan en ocasiones con información de lapso temporal distribuida a lo largo de varias columnas. Por ejemplo, en este conjunto de datos macroeconómico, el año y el trimestre están en columnas distintas:

```
In [200]: data = pd.read_csv("examples/macrodata.csv")
```

```
In [201]: data.head(5)
```

```
Out[201]:
```

	year	quarter	realgdp	realcons	realinv	realgovt	realdpi	cpi	\
0	1959	1	2710.349	1707.4	286.898	470.045	1886.9	28.98	
1	1959	2	2778.801	1733.7	310.859	481.301	1919.7	29.15	
2	1959	3	2775.488	1751.8	289.226	491.260	1916.4	29.35	
3	1959	4	2785.204	1753.7	299.356	484.052	1931.3	29.37	
4	1960	1	2847.699	1770.5	331.722	462.199	1955.5	29.54	
		m1	tbilrate	unemp	pop	infl	realint		
0	139.7	2.82	5.8	177.146	0.00	0.00			
1	141.7	3.08	5.1	177.830	2.34	0.74			
2	140.5	3.82	5.3	178.657	2.74	1.09			
3	140.0	4.33	5.6	179.386	0.27	4.06			
4	139.6	3.50	5.2	180.007	2.31	1.19			

```
In [202]: data["year"]
```

```
Out[202]:
```

0	1959
1	1959
2	1959
3	1959
4	1960
	...

```
198          2008
199          2008
200          2009
201          2009
202          2009
```

Name: year, Length: 203, dtype: int64

```
In [203]: data["quarter"]
Out[203]:
```

```
0            1
1            2
2            3
3            4
4            1
...
198          3
199          4
200          1
201          2
202          3
```

Name: quarter, Length: 203, dtype: int64

Pasando estos arrays a PeriodIndex con una frecuencia, se pueden combinar para formar un índice para el dataframe:

```
In [204]: index = pd.PeriodIndex(year=data["year"],
quarter=data["quarter"],
```

```
.....:           freq="Q-DEC")
```

```
In [205]: index
Out[205]:
```

```
PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4', '1960Q1',
'1960Q2',
'1960Q3', '1960Q4', '1961Q1', '1961Q2',
...
'2007Q2', '2007Q3', '2007Q4', '2008Q1', '2008Q2', '2008Q3',
```

```
'2008Q4', '2009Q1', '2009Q2', '2009Q3'],
dtype='period[Q-DEC]', length=203)
```

```
In [206]: data.index = index
```

```
In [207]: data["infl"]
```

```
Out[207]:
```

1959Q1	0.00
1959Q2	2.34
1959Q3	2.74
1959Q4	0.27
1960Q1	2.31
...	...
2008Q3	-3.16
2008Q4	-8.79
2009Q1	0.94
2009Q2	3.37
2009Q3	3.56

```
Freq: Q-DEC, Name: infl, Length: 203, dtype: float64
```

11.6 Remuestreo y conversión de frecuencias

El término remuestreo se refiere al proceso de convertir una serie temporal de una frecuencia a otra. Al proceso de agregar datos de alta frecuencia a baja frecuencia se le denomina submuestreo, mientras que al proceso contrario se le llama sobremuestreo. Pero no todos los remuestreos entran en estas dos categorías; por ejemplo, convertir w-WED (semanalmente los miércoles) en w-FRI no es ni sobremuestreo ni submuestreo.

Los objetos pandas van equipados con un método `resample`, la función principal para todos los tipos de conversión de frecuencias. Este método tiene una API similar a `groupby`; primero llamamos a `resample` para agrupar los datos, y después llamamos a una función de agregación:

```
In [208]: dates = pd.date_range("2000-01-01", periods=100)
```

```
In [209]: ts = pd.Series(np.random.standard_normal(len(dates)),
index=dates)
```

```
In [210]: ts
```

```
Out[210]:
```

2000-01-01	0.631634
2000-01-02	-1.594313
2000-01-03	-1.519937
2000-01-04	1.108752
2000-01-05	1.255853
	...
2000-04-05	-0.423776
2000-04-06	0.789740
2000-04-07	0.937568
2000-04-08	-2.253294
2000-04-09	-1.772919

Freq: D, Length: 100, dtype: float64

```
In [211]: ts.resample("M").mean()  
Out[211]:
```

2000-01-31	-0.165893
2000-02-29	0.078606
2000-03-31	0.223811
2000-04-30	-0.063643

Freq: M, dtype: float64

```
In [212]: ts.resample("M", kind="period").mean()  
Out[212]:
```

2000-01	-0.165893
2000-02	0.078606
2000-03	0.223811
2000-04	-0.063643

Freq: M, dtype: float64

El método `resample` es flexible y puede utilizarse para procesar grandes series temporales. Los ejemplos de las siguientes secciones ilustran su semántica y uso. La tabla 11.5 resume algunas de sus opciones.

Tabla 11.5. Argumentos del método resample.

Argumento	Descripción
rule	Cadena de texto, DateOffset o timedelta que indica la frecuencia de remuestreo deseada (por ejemplo, "M", "5min" o Second(15)).
axis	Eje según el cual remuestrear; por defecto es axis=0.
fill_method	Cómo interpolar al sobremuestrear, como en "ffill" o "bfill"; de forma predeterminada no hace interpolación.
closed	En submuestreos, qué extremo de cada intervalo está cerrado (inclusive), "right" o "left".
label	En submuestreos, cómo etiquetar el resultado agregado, con el borde de contenedor "right" o "left" (por ejemplo, el intervalo de cinco minutos de 9:30 a 9:35 se puede etiquetar como 9:30 o 9:35).
limit	Cuando se rellena hacia delante o hacia atrás, el número máximo de periodos que se van a llenar.
kind	Agrega a periodos ("period") o marcas temporales ("timestamp"); el valor predeterminado es el tipo de índice que tiene la serie temporal.
convention	Cuando se remuestrean periodos, el convenio ("start" o "end") para convertir el periodo de baja frecuencia en alta frecuencia; por defecto es "start".
origin	La marca temporal «base» a partir de la cual se determinan los bordes de contenedor de remuestreo; también puede ser "epoch", "start", "start_day", "end" o "end_day". En el docstring de resample se dispone de más información.
offset	Un timedelta de desfase añadido al origen; su valor predeterminado es None.

Submuestreo

Submuestrear es agregar datos a una frecuencia baja regular. Los datos que estamos agregando no tienen que ser fijados con frecuencia; la frecuencia deseada define los bordes del contenedor que se utilizan para segmentar la serie temporal en los fragmentos que se van a agregar. Por ejemplo, para convertir a mensual, "M" o "BM", basta con dividir los datos en intervalos de un mes. Se dice que cada intervalo es mitad abierto; un punto de datos puede pertenecer solamente a un intervalo, y la unión de los intervalos debe formar el marco de tiempo completo. Hay un par de cosas en las que conviene pensar cuando se emplea resample para submuestrear datos:

- Qué lado de cada intervalo está cerrado.
- Cómo etiquetar cada contenedor agregado, ya sea con el inicio del intervalo o con el final.

A modo de ilustración, veamos unos datos de frecuencia de un minuto:

```
In [213]: dates = pd.date_range("2000-01-01", periods=12, freq="T")
```

```
In [214]: ts = pd.Series(np.arange(len(dates)), index=dates)
```

```
In [215]: ts
Out[215]:
```

2000-01-01	00:00:00	0
2000-01-01	00:01:00	1
2000-01-01	00:02:00	2
2000-01-01	00:03:00	3
2000-01-01	00:04:00	4
2000-01-01	00:05:00	5
2000-01-01	00:06:00	6
2000-01-01	00:07:00	7
2000-01-01	00:08:00	8
2000-01-01	00:09:00	9
2000-01-01	00:10:00	10
2000-01-01	00:11:00	11

Freq: T, dtype: int64

Supongamos que queremos agregar estos datos en fragmentos o barras de cinco minutos tomando la suma de cada grupo:

```
In [216]: ts.resample("5min").sum()
Out[216]:
```

2000-01-01	00:00:00	10
2000-01-01	00:05:00	35
2000-01-01	00:10:00	21

Freq: 5T, dtype: int64

La frecuencia que se pasa define los bordes del contenedor en incrementos de cinco minutos. Para esta frecuencia, el borde izquierdo del contenedor es inclusivo por defecto, de modo que el valor 00:00 está incluido en el intervalo 00:00 a 00:05, y el valor 00:05 está excluido de dicho intervalo².

```
In [217]: ts.resample("5min", closed="right").sum()  
Out[217]:
```

1999-12-31	23:55:00	0
2000-01-01	00:00:00	15
2000-01-01	00:05:00	40
2000-01-01	00:10:00	11

Freq: 5T, dtype: int64

La serie resultante es etiquetada por las marcas temporales desde el lado izquierdo de cada contenedor. Pasando `label="right"` se pueden etiquetar con el borde derecho del contenedor:

```
In [218]: ts.resample("5min", closed="right",  
label="right").sum()  
Out[218]:
```

2000-01-01	00:00:00	0
2000-01-01	00:05:00	15
2000-01-01	00:10:00	40
2000-01-01	00:15:00	11

Freq: 5T, dtype: int64

En la figura 11.3 se muestra una ilustración de datos de frecuencia de minuto que se están remuestreando a una frecuencia de cinco minutos.

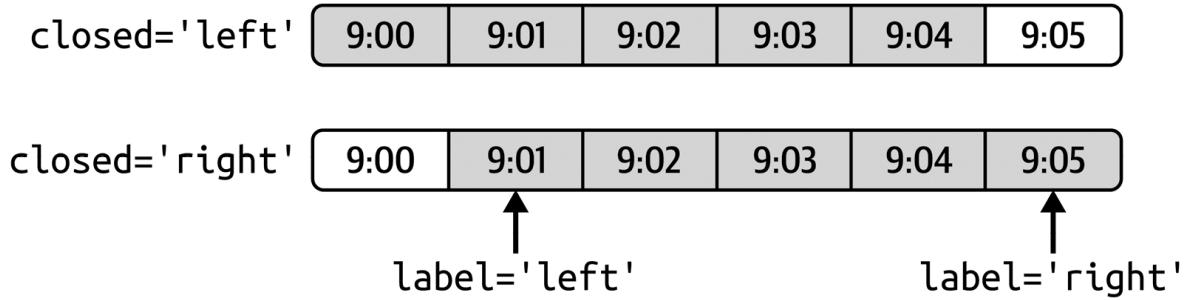


Figura 11.3. Ilustración de remuestreo de cinco minutos con los convenios closed y label.

Por último, puede que queramos desplazar el índice del resultado en una cierta cantidad, digamos restando un segundo desde el contenedor derecho para dejar más claro a qué intervalo se refiere la marca temporal. Para hacer esto, añadimos un desfase al índice resultante:

```
In [219]: from pandas.tseries.frequencies import to_offset
In [220]: result = ts.resample("5min", closed="right",
label="right").sum()
In [221]: result.index = result.index + to_offset("-1s")
In [222]: result
Out[222]:
```

1999-12-31	23:59:59	0
2000-01-01	00:04:59	15
2000-01-01	00:09:59	40
2000-01-01	00:14:59	11

Freq: 5T, dtype: int64

Remuestreo OHLC (Open-high-low-close: abrir-alto-bajo-cerrar)

En finanzas, una forma habitual de agregar una serie temporal es calcular cuatro valores para cada contenedor: el primero (*open*: abrir), el último (*close*: cerrar), el máximo (*high*: alto) y el mínimo (*low*: bajo). Utilizando la función de agregación `ohlc` obtendremos un dataframe cuyas columnas contienen estos cuatro agregados, que se calculan de forma eficaz en una única llamada a función:

```
In [223]: ts = pd.Series(np.random.permutation(np.arange(len(dates))), index=dates)
```

```
In [224]: ts.resample("5min").ohlc()
```

```
Out[224]:
```

		open	high	low	close
2000-01-01	00:00:00	8	8	1	5
2000-01-01	00:05:00	6	11	2	2
2000-01-01	00:10:00	0	7	0	7

Sobremuestreo e interpolación

Sobremuestrear es convertir de una frecuencia baja a otra alta, donde no se necesita agregación. Veamos un dataframe con algunos datos semanales:

```
In [225]: frame = pd.DataFrame(np.random.standard_normal((2, 4)),
```

```
.....:     index=pd.date_range("2000-01-01", periods=2,
```

```
.....:     freq="W-WED"),
```

```
.....:     columns=[“Colorado”, “Texas”, “New York”, “Ohio”])
```

```
In [226]: frame
```

```
Out[226]:
```

	Colorado	Texas	New York	Ohio
2000-01-05	-0.896431	0.927238	0.482284	-0.867130
2000-01-12	0.493841	-0.155434	1.397286	1.507055

Cuando utilizamos una función de agregación con estos datos, solo hay un valor por grupo, y aparecen valores ausentes en los huecos. Empleamos el método `asfreq` para convertir a la frecuencia más alta sin agregación alguna:

```
In [227]: df_daily = frame.resample("D").asfreq()
```

```
In [228]: df_daily
```

```
Out[228]:
```

Colorado	Texas	New York	Ohio
----------	-------	----------	------

2000-01-05	-0.896431	0.927238	0.482284	-0.867130
2000-01-06	NaN	NaN	NaN	NaN
2000-01-07	NaN	NaN	NaN	NaN
2000-01-08	NaN	NaN	NaN	NaN
2000-01-09	NaN	NaN	NaN	NaN
2000-01-10	NaN	NaN	NaN	NaN
2000-01-11	NaN	NaN	NaN	NaN
2000-01-12	0.493841	-0.155434	1.397286	1.507055

Supongamos que queremos rellenar hacia adelante todos los valores semanales que no sean miércoles. Los mismos métodos de relleno o interpolación que están disponibles en los métodos `fillna` y `reindex` lo están para remuestrear:

```
In [229]: frame.resample("D").ffill()
Out[229]:
```

	Colorado	Texas	NewYork	Ohio
2000-01-05	-0.896431	0.927238	0.482284	-0.867130
2000-01-06	-0.896431	0.927238	0.482284	-0.867130
2000-01-07	-0.896431	0.927238	0.482284	-0.867130
2000-01-08	-0.896431	0.927238	0.482284	-0.867130
2000-01-09	-0.896431	0.927238	0.482284	-0.867130
2000-01-10	-0.896431	0.927238	0.482284	-0.867130
2000-01-11	-0.896431	0.927238	0.482284	-0.867130
2000-01-12	0.493841	-0.155434	1.397286	1.507055

De forma similar, se puede elegir rellenar solamente un determinado número de períodos hacia delante para limitar hasta dónde continuar utilizando un valor observado:

```
In [230]: frame.resample("D").ffill(limit=2)
Out[230]:
```

	Colorado	Texas	NewYork	Ohio
2000-01-05	-0.896431	0.927238	0.482284	-0.867130
2000-01-06	-0.896431	0.927238	0.482284	-0.867130
2000-01-07	-0.896431	0.927238	0.482284	-0.867130
2000-01-08	NaN	NaN	NaN	NaN
2000-01-09	NaN	NaN	NaN	NaN
2000-01-10	NaN	NaN	NaN	NaN

2000-01-11	NaN	NaN	NaN	NaN
2000-01-12	0.493841	-0.155434	1.397286	1.507055

Particularmente se observa que el nuevo índice de fecha no tiene por qué coincidir con el antiguo:

```
In [231]: frame.resample("W-THU").ffill()
Out[231]:
```

	Colorado	Texas	NewYork	Ohio
2000-01-06	-0.896431	0.927238	0.482284	-0.867130
2000-01-13	0.493841	-0.155434	1.397286	1.507055

Remuestreo con periodos

El remuestreo de datos indexados por periodos es similar a las marcas de tiempo:

```
In [232]: frame = pd.DataFrame(np.random.standard_normal((24,
4)),
.....:     index=pd.period_range("1-2000", "12-2001",
.....:     freq="M"),
.....:     columns=["Colorado", "Texas", "New York", "Ohio"])
```

```
In [233]: frame.head()
Out[233]:
```

	Colorado	Texas	NewYork	Ohio
2000-01	-1.179442	0.443171	1.395676	-0.529658
2000-02	0.787358	0.248845	0.743239	1.267746
2000-03	1.302395	-0.272154	-0.051532	-0.467740
2000-04	-1.040816	0.426419	0.312945	-1.115689
2000-05	1.234297	-1.893094	-1.661605	-0.005477

```
In [234]: annual_frame = frame.resample("A-DEC").mean()
```

```
In [235]: annual_frame
Out[235]:
```

	Colorado	Texas	NewYork	Ohio
--	----------	-------	---------	------

2000	0.487329	0.104466	0.020495	-0.273945
2001	0.203125	0.162429	0.056146	-0.103794

El sobremuestreo está más matizado, porque antes de muestrear es necesario tomar una decisión sobre el extremo del lapso de tiempo de la nueva frecuencia en el que poner los valores. El argumento `convention` tiene como valor predeterminado “start” pero también puede ser “end”:

```
# Q-DEC: Trimestral, el año termina en diciembre
```

```
In [236]: annual_frame.resample("Q-DEC").ffill()
Out[236]:
```

	Colorado	Texas	NewYork	Ohio
2000Q1	0.487329	0.104466	0.020495	-0.273945
2000Q2	0.487329	0.104466	0.020495	-0.273945
2000Q3	0.487329	0.104466	0.020495	-0.273945
2000Q4	0.487329	0.104466	0.020495	-0.273945
2001Q1	0.203125	0.162429	0.056146	-0.103794
2001Q2	0.203125	0.162429	0.056146	-0.103794
2001Q3	0.203125	0.162429	0.056146	-0.103794
2001Q4	0.203125	0.162429	0.056146	-0.103794

```
In [237]: annual_frame.resample("Q-DEC",
convention="end").asfreq()
Out[237]:
```

	Colorado	Texas	NewYork	Ohio
2000Q4	0.487329	0.104466	0.020495	-0.273945
2001Q1	NaN	NaN	NaN	NaN
2001Q2	NaN	NaN	NaN	NaN
2001Q3	NaN	NaN	NaN	NaN
2001Q4	0.203125	0.162429	0.056146	-0.103794

Como los periodos se refieren a lapsos de tiempo, las reglas del sobremuestreo y submuestreo son más rígidas:

- En el submuestreo, la frecuencia de destino debe ser un subperiodo de la frecuencia de origen.

- En el sobremuestreo, la frecuencia de destino debe ser un superperiodo de la frecuencia de origen.

Si estas reglas no se cumplen, se puede producir un error. Esto afecta principalmente a las frecuencias trimestral, anual y semanal; por ejemplo, los lapsos de tiempo definidos por Q-MAR solo se alinean con A-MAR, A-JUN, A-SEP y A-DEC:

```
In [238]: annual_frame.resample("Q-MAR").ffill()
Out[238]:
```

	Colorado	Texas	NewYork	Ohio
2000Q4	0.487329	0.104466	0.020495	-0.273945
2001Q1	0.487329	0.104466	0.020495	-0.273945
2001Q2	0.487329	0.104466	0.020495	-0.273945
2001Q3	0.487329	0.104466	0.020495	-0.273945
2001Q4	0.203125	0.162429	0.056146	-0.103794
2002Q1	0.203125	0.162429	0.056146	-0.103794
2002Q2	0.203125	0.162429	0.056146	-0.103794
2002Q3	0.203125	0.162429	0.056146	-0.103794

Remuestreo de tiempo agrupado

Para datos de series temporales, el método `resample` es semánticamente una operación de grupo basada en una intervalización de tiempo. Aquí tenemos una tabla de ejemplo:

```
In [239]: N = 15
```

```
In [240]: times = pd.date_range("2017-05-20 00:00", freq="1min",
                               periods=N)
```

```
In [241]: df = pd.DataFrame({"time": times,
                           ....: "value": np.arange(N)})
```

```
In [242]: df
```

```
Out[242]:
```

		time	value
0		2017-05-20 00:00:00	0
1		2017-05-20 00:01:00	1
2		2017-05-20 00:02:00	2

3	2017-05-20	00:03:00	3
4	2017-05-20	00:04:00	4
5	2017-05-20	00:05:00	5
6	2017-05-20	00:06:00	6
7	2017-05-20	00:07:00	7
8	2017-05-20	00:08:00	8
9	2017-05-20	00:09:00	9
10	2017-05-20	00:10:00	10
11	2017-05-20	00:11:00	11
12	2017-05-20	00:12:00	12
13	2017-05-20	00:13:00	13
14	2017-05-20	00:14:00	14

Aquí, podemos indexar por “time” y después remuestrear:

```
In [243]: df.set_index("time").resample("5min").count()
Out[243]:
```

time	value	
2017-05-20	00:00:00	5
2017-05-20	00:05:00	5
2017-05-20	00:10:00	5

Supongamos que un dataframe contiene varias series temporales, marcadas por una columna de clave de grupo adicional:

```
In [244]: df2 = pd.DataFrame({"time": times.repeat(3),
....: "key": np.tile(["a", "b", "c"], N),
....: "value": np.arange(N * 3.)})
```

```
In [245]: df2.head(7)
Out[245]:
```

		time	key	value
0		2017-05-20	a	0.0
1		2017-05-20	b	1.0
2		2017-05-20	c	2.0
3		2017-05-20	a	3.0

```

4          2017-05-20      00:01:00      b      4.0
5          2017-05-20      00:01:00      c      5.0
6          2017-05-20      00:02:00      a      6.0

```

Para hacer el mismo remuestreo para cada valor de “key”, introducimos el objeto pandas.Grouper:

```
In [246]: time_key = pd.Grouper(freq="5min")
```

Podemos a continuación establecer el índice de hora, agrupar por “key” y time_key, y finalmente agregar:

```
In [247]: resampled = (df2.set_index("time")
```

```
.....:     .groupby(["key", time_key])
.....:     .sum())
```

```
In [248]: resampled
```

```
Out[248]:
```

			value
key		time	
a	2017-05-20	00:00:00	30.0
	2017-05-20	00:05:00	105.0
	2017-05-20	00:10:00	180.0
b	2017-05-20	00:00:00	35.0
	2017-05-20	00:05:00	110.0
	2017-05-20	00:10:00	185.0
c	2017-05-20	00:00:00	40.0
	2017-05-20	00:05:00	115.0
	2017-05-20	00:10:00	190.0

```
In [249]: resampled.reset_index()
```

```
Out[249]:
```

		key	time	value
0	a	2017-05-20	00:00:00	30.0
1	a	2017-05-20	00:05:00	105.0
2	a	2017-05-20	00:10:00	180.0
3	b	2017-05-20	00:00:00	35.0
4	b	2017-05-20	00:05:00	110.0

5	b	2017-05-20	00:10:00	185.0
6	c	2017-05-20	00:00:00	40.0
7	c	2017-05-20	00:05:00	115.0
8	c	2017-05-20	00:10:00	190.0

Una limitación del uso de pandas.Grouper es que la hora debe ser el índice del objeto Series o DataFrame.

11.7 Funciones de ventana móvil

Una clase importante de transformaciones de array empleadas para operaciones con series temporales son estadísticas y otras funciones evaluadas a lo largo de una ventana deslizante o con pesos que se descomponen exponencialmente. Esto puede resultar útil para suavizar datos con mucho ruido o muchos huecos. Yo las llamo funciones de ventana móvil, incluso aunque incluyan funciones sin ventana de longitud fija, como el promedio móvil ponderado exponencialmente. Al igual que otras funciones estadísticas, estas también excluyen de forma automática los datos faltantes.

Antes de meternos a fondo, podemos cargar algunos datos de series temporales y remuestrearlos a una frecuencia de día laborable:

```
In [250]: close_px_all = pd.read_csv("examples/stock_px.csv",
.....:           parse_dates=True, index_col=0)

In [251]: close_px = close_px_all[["AAPL", "MSFT", "XOM"]]

In [252]: close_px = close_px.resample("B").ffill()
```

Ahora introduzco el operador `rolling`, que se comporta de forma similar a `resample` y `groupby`. Se le puede llamar en un objeto Series o DataFrame junto con un operador `window` (expresado como un número de períodos; véase en la figura 11.4 el gráfico que se ha creado):

```
In [253]: close_px["AAPL"].plot()
Out[253]: <AxesSubplot:>

In [254]: close_px["AAPL"].rolling(250).mean().plot()
```

La expresión `rolling(250)` es similar en comportamiento a `groupby`, pero en lugar de agrupar sin más, crea un objeto que permite agrupar sobre una ventana deslizante de 250 días. Así, aquí tenemos el promedio de ventana móvil de la cotización en bolsa de Apple.

De forma predeterminada, las funciones rodantes requieren que todos los valores de la ventana sean no nulos o no NA. Este comportamiento puede cambiarse para tener en cuenta los datos ausentes y, en particular, el hecho de que tendremos menos periodos de datos window al principio de la serie temporal (véase la figura 11.5):



Figura 11.4. Precio de Apple con un promedio móvil de 250 días.

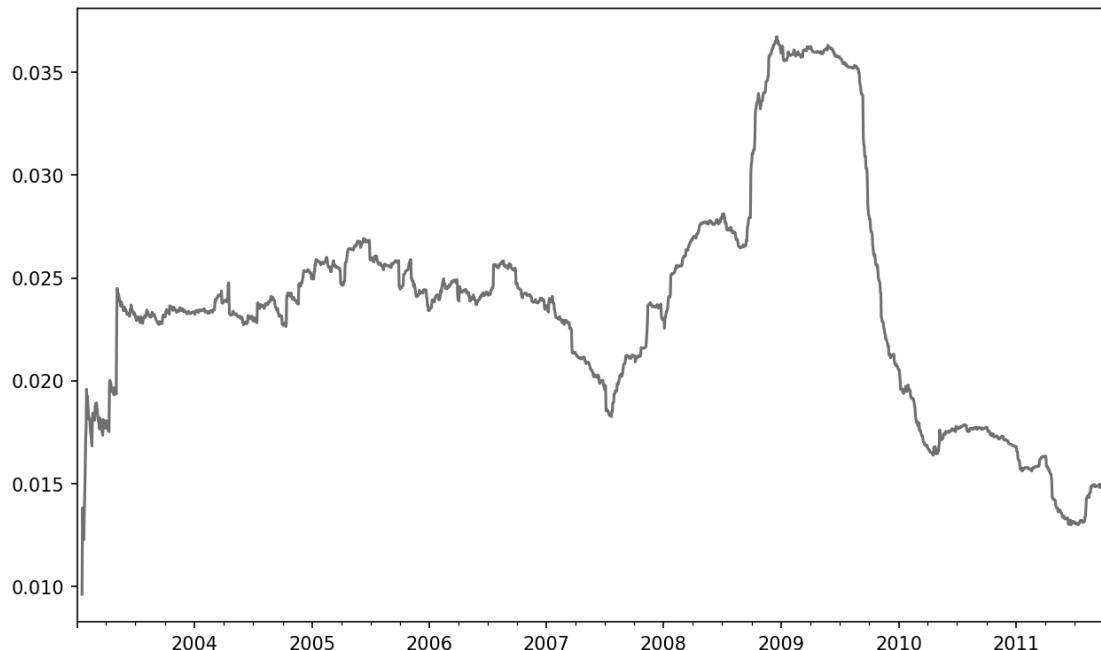


Figura 11.5. Desviación estándar de la rentabilidad diaria en 250 días.

```
In [255]: plt.figure()
```

```
Out[255]: <Figure size 1000x600 with 0 Axes>
```

```
In [256]: std250 = close_px["AAPL"].pct_change().rolling(250,
min_periods=10).std()
```

```
In [257]: std250[5:12]
```

```
Out[257]:
```

2003-01-09	NaN
2003-01-10	NaN
2003-01-13	NaN
2003-01-14	NaN
2003-01-15	NaN
2003-01-16	0.009628
2003-01-17	0.013818

```
Freq: B, Name: AAPL, dtype: float64
```

```
In [258]: std250.plot()
```

Para calcular una media de ventana expansiva, utilizamos el operador `expanding` en lugar de `rolling`. La media expansiva inicia la ventana de tiempo

desde el mismo punto que la ventana rodante y aumenta el tamaño de la ventana hasta que cubre toda la serie. Una media de ventana expansiva en la serie temporal std250 tiene un aspecto similar a este:

```
In [259]: expanding_mean = std250.expanding().mean()
```

Llamar a una función de ventana en movimiento en un dataframe aplica la transformación a cada columna (véase la figura 11.6):

```
In [261]: plt.style.use('grayscale')
```

```
In [262]: close_px.rolling(60).mean().plot(logy=True)
```

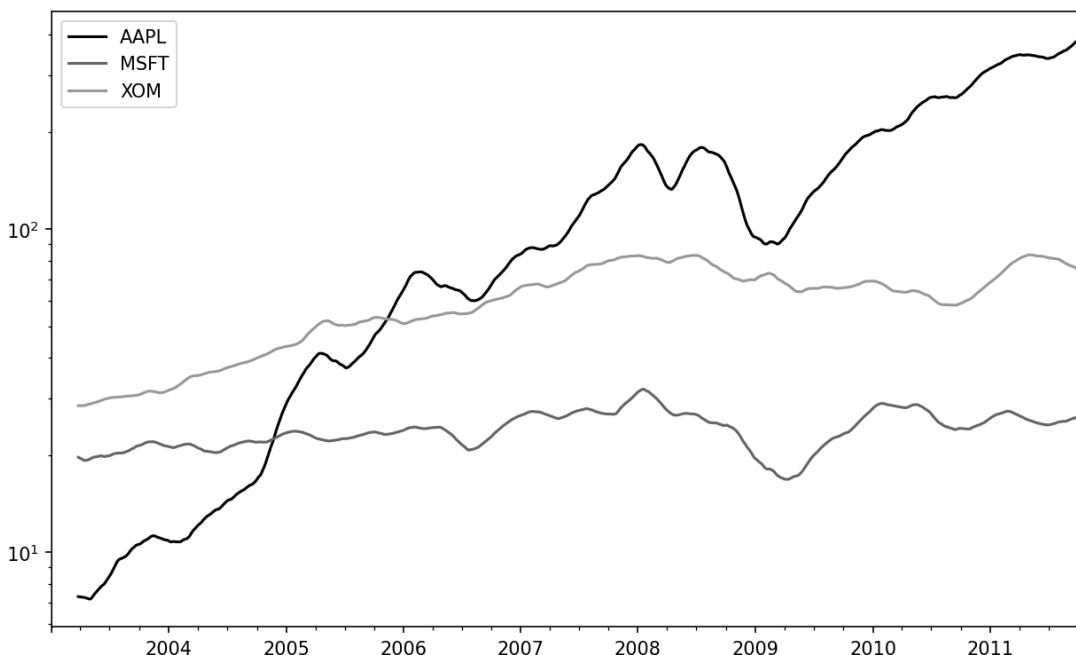


Figura 11.6. Promedio móvil de 60 días de la cotización en bolsa (logaritmo del eje y).

La función `rolling` acepta asimismo una cadena de texto indicando un desplazamiento de tiempo de tamaño fijo `rolling()` en funciones de ventana móvil en vez de un número fijo de períodos. Emplear esta notación puede ser útil para series temporales irregulares. Son las mismas cadenas de texto que se le pueden pasar a `resample`. Por ejemplo, podríamos calcular una media rodante de 20 días de esta forma:

```
In [263]: close_px.rolling("20D").mean()  
Out[263]:
```

	AAPL	MSFT	XOM
2003-01-02	7.400000	21.110000	29.220000
2003-01-03	7.425000	21.125000	29.230000
2003-01-06	7.433333	21.256667	29.473333
2003-01-07	7.432500	21.425000	29.342500
2003-01-08	7.402000	21.402000	29.240000
...
2011-10-10	389.351429	25.602143	72.527857
2011-10-11	388.505000	25.674286	72.835000
2011-10-12	388.531429	25.810000	73.400714
2011-10-13	388.826429	25.961429	73.905000
2011-10-14	391.038000	26.048667	74.185333

[2292 rows x 3 columns]

Funciones ponderadas exponencialmente

Una alternativa a utilizar un tamaño de ventana fijo con observaciones ponderadas por igual es especificar un factor de descomposición constante para dar más peso a observaciones más recientes. Hay un par de maneras de especificar el factor de descomposición. Una muy utilizada es emplear un lapso, que logra que el resultado sea comparable con una sencilla función móvil con un tamaño de ventana igual al lapso.

Como una estadística exponencialmente ponderada pone más peso en observaciones más recientes, se «adapta» más rápido a los cambios, comparada con la versión del mismo peso.

pandas tiene el operador `ewm` (*Exponentially Weighted Moving*: móvil y exponencialmente ponderado) para aceptar a la vez `rolling` y `expanding`. Aquí tenemos un ejemplo que compara un promedio móvil de 30 días de la cotización en bolsa de Apple con un promedio móvil exponencialmente ponderado (EW: *Exponentially Weighted*) con `span=60` (figura 11.7):

```
In [265]: aapl_px = close_px["AAPL"]["2006":"2007"]
```

```
In [266]: ma30 = aapl_px.rolling(30, min_periods=20).mean()
```

```
In [267]: ewma30 = aapl_px.ewm(span=30).mean()
```

```
In [268]: aapl_px.plot(style="k-", label="Price")
```

```
Out[268]: <AxesSubplot:>
```

```
In [269]: ma30.plot(style="k-", label="Simple Moving Avg")
Out[269]: <AxesSubplot:>

In [270]: ewma30.plot(style="k-", label="EW MA")
Out[270]: <AxesSubplot:>

In [271]: plt.legend()
```

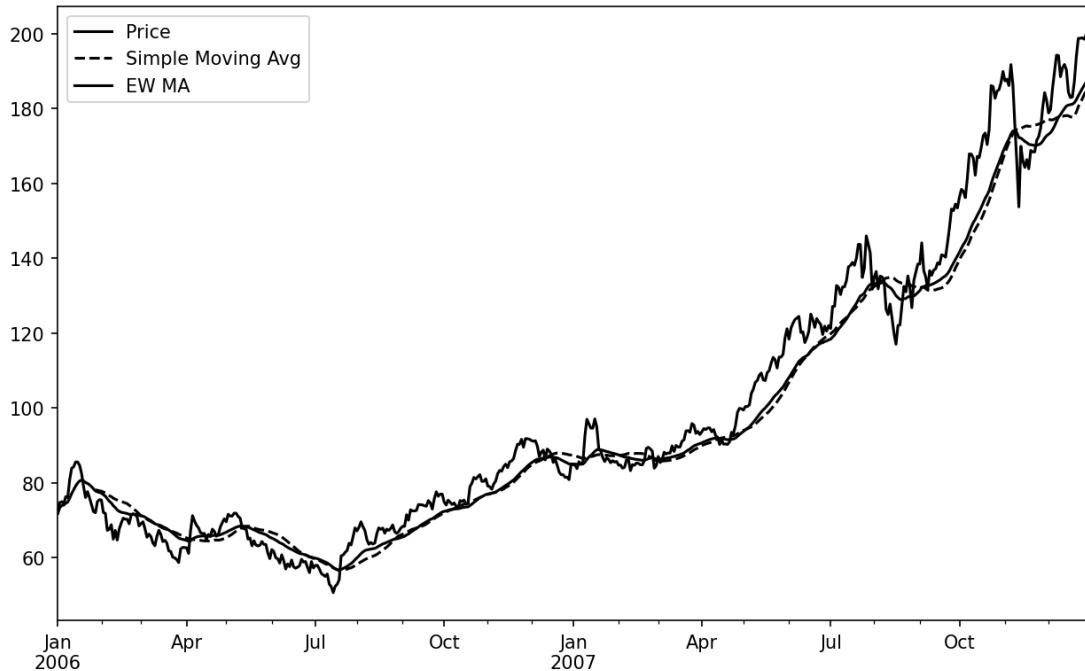


Figura 11.7. Promedio móvil sencillo frente a exponencialmente ponderado.

Funciones binarias de ventana móvil

Algunos operadores estadísticos, como la correlación y la covarianza, tienen que funcionar con dos series temporales. Como ejemplo, los analistas financieros suelen interesarse por la correlación de acciones con respecto a un índice de referencia como S&P 500. Vamos a ver esto calculando primero el cambio de porcentaje para todas nuestras series temporales de interés:

```
In [273]: spx_px = close_px_all["SPX"]
In [274]: spx_rets = spx_px.pct_change()
In [275]: returns = close_px.pct_change()
```

Después de llamar a `rolling`, la función de agregación `corr` puede calcular la correlación rodante con `spx_rets` (en la figura 11.8 puede verse el gráfico resultante):

```
In [276]: corr = returns["AAPL"].rolling(125,  
min_periods=100).corr(spx_rets)  
  
In [277]: corr.plot()
```

Supongamos que queremos calcular la correlación rodante del índice S&P 500 con muchas acciones al mismo tiempo. Se podría escribir un bucle que calculara esto para cada acción como hicimos antes para Apple, pero si cada acción es una columna de un dataframe, podemos calcular todas las correlaciones rodantes de una vez llamando a `rolling` sobre el objeto DataFrame y pasando la serie `spx_rets`.



Figura 11.8. Correlación de la rentabilidad de AAPL en seis meses con respecto a S&P 500.

En la figura 11.9 se puede contemplar el gráfico del resultado:

```
In [279]: corr = returns.rolling(125,  
min_periods=100).corr(spx_rets)  
  
In [280]: corr.plot()
```

Funciones de ventana móvil definidas por el usuario

El método `apply` sobre `rolling` y otros métodos asociados ofrecen una forma de aplicar una función de array creada por uno mismo a una ventana en movimiento. El único requisito es que la función produzca un solo valor (una reducción) de cada parte del array. Por ejemplo, aunque podemos calcular cuantiles de muestra utilizando `rolling(...).quantile(q)`, quizás nos interesaría más el rango de percentil de un determinado valor sobre la muestra. La función `scipy.stats.percentileofscore` hace justamente esto (en la figura 11.10 se muestra el gráfico resultante):

```
In [282]: from scipy.stats import percentileofscore
```

```
In [283]: def score_at_2percent(x):
```

```
....:     return percentileofscore(x, 0.02)
```

```
In [284]: result =  
returns["AAPL"].rolling(250).apply(score_at_2percent)
```

```
In [285]: result.plot()
```

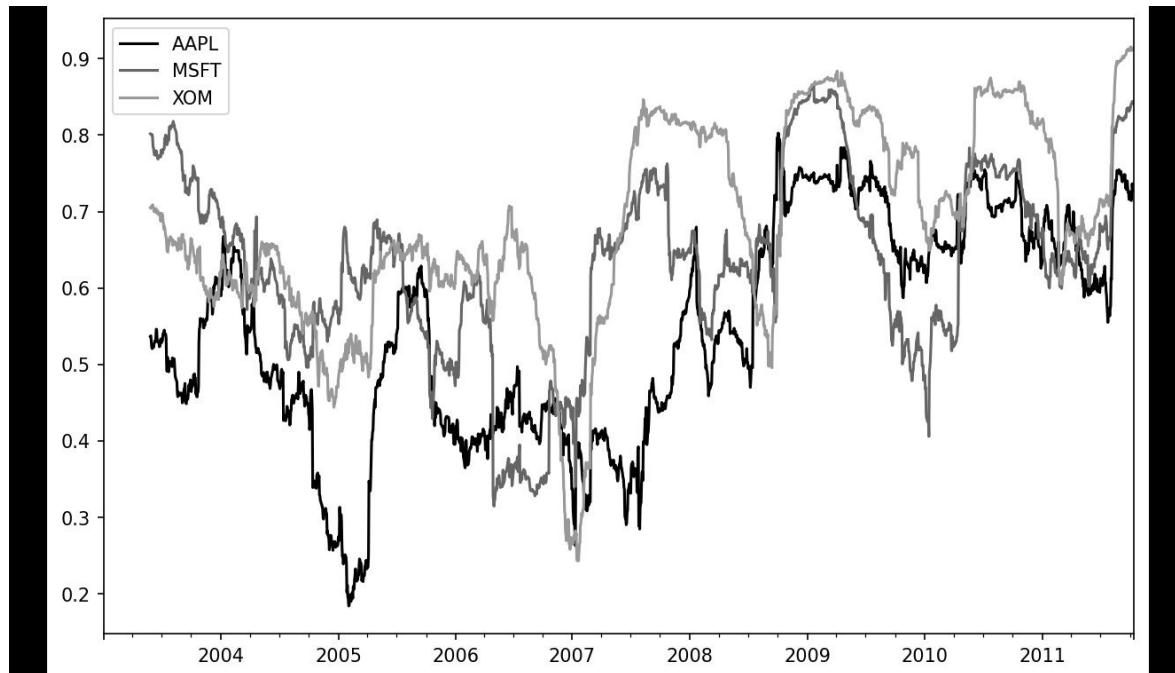


Figura 11.9. Correlaciones de la rentabilidad en seis meses con respecto a S&P 500.



Figura 11.10. Rango de percentil de un rendimiento de AAPL del 2 % sobre una ventana de un año.

Si no está SciPy ya instalado, se puede instalar con conda o pip:

```
conda install scipy
```

11.8 Conclusión

Los datos de series temporales requieren unos tipos de análisis y herramientas de transformación de datos distintos a otros tipos de datos que ya hemos explorado en capítulos anteriores.

En el siguiente veremos cómo empezar a utilizar librerías de modelado como statsmodels y scikit-learn.

² La elección de los valores predeterminados para `closed` y `label` puede parecerles un poco extraña a algunos usuarios. El valor predeterminado es `closed="left"` para todo, excepto para un determinado conjunto ("M", "A", "Q", "BM", "BQ" y "W") para el que es `closed="right"`. Estos valores se eligieron para que los resultados fueran más intuitivos, pero conviene saber que el valor por defecto no es siempre uno u otro.

Introducción a las librerías de creación de modelos de Python

En este libro me he centrado en ofrecer una base de programación para realizar análisis de datos en Python. Como los analistas y científicos de datos suelen informar del uso de una cantidad de tiempo desproporcionada en la disputa y preparación de los datos, la estructura del libro refleja la importancia de dominar estas técnicas.

La librería que se utilice para desarrollar modelos dependerá de la aplicación. Muchos problemas estadísticos pueden resolverse con técnicas sencillas, como la regresión de mínimos cuadrados ordinarios, mientras que otros problemas pueden requerir métodos de aprendizaje automático más avanzados. Por suerte, Python se ha convertido en uno de los lenguajes preferidos para implementar métodos analíticos, de modo que hay muchas herramientas por explorar una vez finalizado este libro.

En este capítulo abordaremos algunas características de pandas que pueden resultar útiles cuando se está cambiando continuamente entre disputa de datos con pandas y ajuste de modelos. Por tanto, ofreceré una breve introducción a cada uno de los dos juegos de herramientas de creación de modelos más conocidos: statsmodels (<http://statsmodels.org>) y scikit-learn (<http://scikit-learn.org>). Como cada uno de estos proyectos es lo bastante grande como para merecer un libro propio, no me esforzaré por ofrecer información detallada, sino que más bien dirigiré a mis lectores a la documentación en línea de ambos proyectos, además de indicar algunos libros basados en Python sobre ciencia de datos, estadísticas y aprendizaje automático.

12.1 Interconexión entre pandas y el código para la creación de modelos

Un flujo de trabajo habitual para el desarrollo de modelos es utilizar pandas para la carga y limpieza de los datos antes de cambiar a una librería específica para crear el modelo como tal. En aprendizaje automático, una parte importante del proceso de creación de modelos se denomina ingeniería de características. Tal denominación describe cualquier transformación o analítica de datos que extrae información de un conjunto de datos sin procesar y que puede ser útil en un contexto de este tipo. La agregación de datos y las herramientas de GroupBy que hemos explorado en este libro se suelen usar en un contexto de ingeniería de características.

Aunque los detalles de una «buena» ingeniería de características quedan fuera del alcance de este libro, mostraré algunos métodos para que el cambio entre la manipulación de datos con pandas y la creación de modelos sea lo más llevadero posible.

El punto de contacto entre pandas y otras librerías de análisis suelen ser los arrays NumPy. Para convertir un objeto DataFrame en un array NumPy, empleamos el método `to_numpy`:

```
In [12]: data = pd.DataFrame({  
....:     'x0': [1, 2, 3, 4, 5],  
....:     'x1': [0.01, -0.01, 0.25, -4.1, 0.],  
....:     'y': [-1.5, 0., 3.6, 1.3, -2.]})
```

```
In [13]: data  
Out[13]:
```

	x0	x1	y
0	1	0.01	-1.5
1	2	-0.01	0.0
2	3	0.25	3.6
3	4	-4.10	1.3
4	5	0.00	-2.0

```
In [14]: data.columns  
Out[14]: Index(['x0', 'x1', 'y'], dtype='object')
```

```
In [15]: data.to_numpy()  
Out[15]:
```

```
array([ [ 1. ,  0.01, -1.5 ],  
       [ 2. , -0.01,  0. ],  
       [ 3. ,  0.25,  3.6 ],  
       [ 4. , -4.1 ,  1.3 ],  
       [ 5. ,  0. , -2. ]])
```

Para convertir de nuevo en un dataframe, como recordamos de anteriores capítulos, podemos pasar un ndarray bidimensional con nombres de columna opcionales:

```
In [16]: df2 = pd.DataFrame(data.to_numpy(), columns=['one',  
'two', 'three'])
```

```
In [17]: df2  
Out[17]:
```

	one	two	three
0	1.0	0.01	-1.5
1	2.0	-0.01	0.0
2	3.0	0.25	3.6
3	4.0	-4.10	1.3
4	5.0	0.00	-2.0

El método `to_numpy` está pensado para ser utilizado cuando los datos son homogéneos, por ejemplo, todos los tipos numéricos. Si tenemos datos heterogéneos, el resultado será un ndarray de objetos Python:

```
In [18]: df3 = data.copy()
```

```
In [19]: df3['strings'] = ['a', 'b', 'c', 'd', 'e']
```

```
In [20]: df3  
Out[20]:
```

	x0	x1	y	strings
0	1	0.01	-1.5	a
1	2	-0.01	0.0	b
2	3	0.25	3.6	c
3	4	-4.10	1.3	d
4	5	0.00	-2.0	e

In [21]: df3.to_numpy()

Out[21]:

```
array([
    [1, 0.01, -1.5, 'a'],
    [2, -0.01, 0.0, 'b'],
    [3, 0.25, 3.6, 'c'],
    [4, -4.1, 1.3, 'd'],
    [5, 0.0, -2.0, 'e']], dtype=object)
```

Para algunos modelos, quizá nos interese emplear únicamente un subconjunto de las columnas. Yo recomiendo utilizar el indexado loc con to_numpy:

In [22]: model_cols = ['x0', 'x1']

In [23]: data.loc[:, model_cols].to_numpy()

Out[23]:

```
array([
    [ 1. , 0.01],
    [ 2. , -0.01],
    [ 3. , 0.25],
    [ 4. , -4.1 ],
    [ 5. , 0. ]])
```

Algunas librerías soportan pandas de forma nativa, por lo que nos hacen parte del trabajo automáticamente: convertir a NumPy desde el objeto DataFrame y adjuntar los nombres de parámetros del modelo a las columnas de las tablas o las series resultantes. En otros casos, tendremos que realizar esta «gestión de metadatos» de forma manual.

En la sección 7.5 «Datos categóricos» vimos el tipo Categorical de pandas y la función pandas.get_dummies. Supongamos que tenemos una columna no numérica en nuestro conjunto de datos de ejemplo:

```
In [24]: data['category'] = pd.Categorical(['a', 'b', 'a',  
'a', 'b'],
```

```
....:           categories=['a', 'b'])
```

```
In [25]: data
```

```
Out[25]:
```

	x0	x1	y	category
0	1	0.01	-1.5	a
1	2	-0.01	0.0	b
2	3	0.25	3.6	a
3	4	-4.10	1.3	a
4	5	0.00	-2.0	b

Si quisiéramos reemplazar la columna ‘category’ con variables indicadoras, crearíamos dichas variables, quitaríamos la columna ‘category’ y después uniríamos el resultado:

```
In [26]: dummies = pd.get_dummies(data.category,  
prefix='category')
```

```
In [27]: data_with_dummies = data.drop('category',  
axis=1).join(dummies)
```

```
In [28]: data_with_dummies
```

```
Out[28]:
```

	x0	x1	y	category_a	category_b
0	1	0.01	-1.5	1	0
1	2	-0.01	0.0	0	1
2	3	0.25	3.6	1	0
3	4	-4.10	1.3	1	0
4	5	0.00	-2.0	0	1

Existen algunos matices a la hora de ajustar determinados modelos estadísticos con variables indicadoras. Quizá lo más sencillo y menos propenso a errores sea utilizar Patsy (el tema de la siguiente sección) cuando tenemos algo más que simples columnas numéricas.

12.2 Creación de descripciones de modelos con Patsy

Patsy (<https://patsy.readthedocs.io/>) es una librería de Python para describir modelos estadísticos (en particular modelos lineales) con una «sintaxis de la fórmula» basada en cadenas de texto, inspirada (aunque no exactamente) por la sintaxis de la fórmula empleada por los lenguajes de programación R y S. Se instala automáticamente con statsmodels:

```
conda install statsmodels
```

Patsy se soporta bien para especificar modelos lineales en statsmodels, de modo que me centraré en algunas de las características principales que permitirán a mis lectores ponerse rápidamente en marcha. Las fórmulas de Patsy tienen una sintaxis de cadena de texto especial que posee este aspecto:

```
y ~ x0 + x1
```

La sintaxis $a + b$ no significa sumar a y b, sino más bien que son términos de la matriz de diseño creada para el modelo. La función `patsy.dmatrices` toma una cadena de texto de fórmula y un conjunto de datos (que puede ser un dataframe o un diccionario de arrays) y produce matrices de diseño para un modelo lineal:

```
In [29]: data = pd.DataFrame({  
....:     'x0': [1, 2, 3, 4, 5],  
....:     'x1': [0.01, -0.01, 0.25, -4.1, 0.],  
....:     'y': [-1.5, 0., 3.6, 1.3, -2.]})
```

```
In [30]: data  
Out[30]:
```

	x0	x1	y
0	1	0.01	-1.5
1	2	-0.01	0.0
2	3	0.25	3.6
3	4	-4.10	1.3
4	5	0.00	-2.0

In [31]: import patsy

In [32]: y, X = patsy.dmatrices('y ~ x0 + x1', data)

Ahora tenemos:

In [33]: y

Out[33]:

DesignMatrix with shape (5, 1)

y

-1.5

0.0

3.6

1.3

-2.0

Terms:

'y' (column 0)

In [34]: X

Out[34]:

DesignMatrix with shape (5, 3)

	Intercept	x0	x1
	1	1	0.01
	1	2	-0.01
	1	3	0.25
	1	4	-4.10
	1	5	0.00

Terms:

'Intercept' (column 0)

'x0' (column 1)

'x1' (column 2)

Estas instancias `DesignMatrix` de Patsy son ndarrays NumPy con metadatos adicionales:

```
In [35]: np.asarray(y)
Out[35]:
```

```
array([[-1.5],
```

```
[ 0. ],
[ 3.6],
[ 1.3],
[-2. ]])
```

```
In [36]: np.asarray(X)
Out[36]:
```

```
array([
      [ 1. ,  1. ,  0.01],
      [ 1. ,  2. , -0.01],
      [ 1. ,  3. ,  0.25],
      [ 1. ,  4. , -4.1 ],
      [ 1. ,  5. ,  0. ]])
```

Es posible que alguno de mis lectores se pregunte de dónde procede el término `Intercept`. Es un convenio para los modelos lineales similar a la regresión de mínimos cuadrados ordinarios (MCO). Se puede suprimir la interceptación añadiendo al modelo el término `+ 0`:

```
In [37]: patsy.dmatrices('y ~ x0 + x1 + 0', data)[1]
Out[37]:
```

```
DesignMatrix with shape (5, 2)
```

x0	x1
1	0.01
2	-0.01
3	0.25
4	-4.10

```
5 0.00
```

Terms:

'x0' (column 0)

'x1' (column 1)

Se pueden pasar objetos Patsy directamente a algoritmos como `numpy.linalg.lstsq`, que realiza una regresión de mínimos cuadrados ordinarios:

```
In [38]: coef, resid, _, _ = np.linalg.lstsq(X, y)
```

Los metadatos del modelo se conservan en el atributo `design_info`, de modo que se puedan volver a adjuntar los nombres de columnas del modelo a los coeficientes ajustados para obtener un objeto Series, por ejemplo:

```
In [39]: coef
```

```
Out[39]:
```

```
array([ 0.3129, -0.0791, -0.2655])
```

```
In [40]: coef = pd.Series(coef.squeeze(), index=X.design_info.column_names)
```

```
In [41]: coef
```

```
Out[41]:
```

```
Intercept      0.312910
x0             -0.079106
x1             -0.265464
```

```
dtype:float64
```

Transformaciones de datos en fórmulas Patsy

Se puede mezclar código Python con las fórmulas Patsy; al evaluar la fórmula, la librería tratará de encontrar las funciones empleadas en el ámbito de aplicación:

```
In [42]: y, X = patsy.dmatrices('y ~ x0 + np.log(np.abs(x1) + 1)', data)
```

```
In [43]: X  
Out[43]:  
DesignMatrix with shape (5, 3)
```

	x0	np.log(np.abs(x1) + 1)
Intercept		
1	1	0.00995
1	2	0.00995
1	3	0.22314
1	4	1.62924
1	5	0.00000

Terms:

```
'Intercept' (column 0)  
'x0' (column 1)  
'np.log(np.abs(x1) + 1)' (column 2)
```

Algunas transformaciones de variables habitualmente utilizadas incluyen la estandarización (a media 0 y varianza 1) y el centrado (restando la media). Patsy tiene funciones integradas para ello:

```
In [44]: y, X = patsy.dmatrices('y ~ standardize(x0) + center(x1)', data)
```

```
In [45]: X  
Out[45]:
```

	standardize(x0)	center(x1)
Intercept		
1	-1.41421	0.78
1	-0.70711	0.76
1	0.00000	1.02

1	0.70711	-3.33
1	1.41421	0.77

Terms:

```
'Intercept' (column 0)
'standardize(x0)' (column 1)
'center(x1)' (column 2)
```

Como parte de un proceso de creación de modelos, es posible ajustar un modelo a un conjunto de datos, para después evaluar el modelo basándose en otro, que podría ser una parte retenida o nuevos datos observados después. Al aplicar transformaciones como centrado y estandarización, conviene tener cuidado al utilizar el modelo para hacer predicciones basadas en datos nuevos. Estas se denominan transformaciones con estado, porque hay que utilizar estadísticas como la media o la desviación estándar del conjunto de datos original al transformar un nuevo conjunto de datos.

La función `patsy.build_design_matrices` puede aplicar transformaciones a nuevos datos de fuera de la muestra utilizando la información guardada del conjunto de datos original de dentro de la muestra:

```
In [46]: new_data = pd.DataFrame({
```

```
....:     'x0': [6, 7, 8, 9],
....:     'x1': [3.1, -0.5, 0, 2.3],
....:     'y': [1, 2, 3, 4]})
```

```
In [47]: new_X = patsy.build_design_matrices([X.design_info], new_data)
```

```
In [48]: new_X
```

```
Out[48]:
```

[DesignMatrix with shape (4, 3)			
Intercept	standardize(x0)	center(x1)	
1	2.12132	3.87	

1	2.82843	0.27
1	3.53553	0.77
1	4.24264	3.07

Terms:

```
'Intercept' (column 0)
'standardize(x0)' (column 1)
'center(x1)' (column 2)]
```

Como el símbolo más (+) del contexto de las fórmulas Patsy no significa suma, cuando queramos sumar columnas desde un conjunto de datos por nombre, tendremos que incluirlas en la función especial `I`:

```
In [49]: y, X = patsy.dmatrices('y ~ I(x0 + x1)', data)
```

```
In [50]: X
```

```
Out[50]:
```

	DesignMatrix with shape (5, 2)
Intercept	$I(x0 + x1)$
1	1.01
1	1.99
1	3.25
1	-0.10
1	5.00

Terms:

```
'Intercept' (column 0)
'I(x0 + x1)' (column 1)
```

Patsy tiene también otras transformaciones integradas en el módulo `patsy.builtins`. La documentación en línea ofrece más información.

Los datos categóricos tienen una clase especial de transformaciones, que explico a continuación.

Datos categóricos y Patsy

Los datos no numéricos se pueden transformar para una matriz de diseño de modelos de muchas formas distintas. Tratar con todo detalle este tema queda fuera del alcance de este libro; sería mucho mejor estudiarlo con un curso de estadística. Al utilizar términos no numéricos en una fórmula Patsy, de forma predeterminada se convierten a variables indicadoras. Si hay una interceptación, uno de los niveles quedará fuera para evitar la colinealidad:

```
In [51]: data = pd.DataFrame({  
    ....:     'key1': ['a', 'a', 'b', 'b', 'a', 'b', 'a',  
    ....:             'b'],  
    ....:     'key2': [0, 1, 0, 1, 0, 1, 0, 0],  
    ....:     'v1': [1, 2, 3, 4, 5, 6, 7, 8],  
    ....:     'v2': [-1, 0, 2.5, -0.5, 4.0, -1.2, 0.2, -1.7]  
    ....:  
})
```

```
In [52]: y, X = patsy.dmatrices('v2 ~ key1', data)
```

```
In [53]: X  
Out[53]:
```

```
DesignMatrix with shape (8, 2)
```

	Intercept	key1[T.b]
1		0
1		0
1		1
1		1
1		0
1		1
1		0
1		1

Terms:

'Intercept' (column 0)

```
'key1' (column 1)
```

Si se omite la interceptación del modelo, entonces las columnas para cada categoría de valor se incluirán en la matriz de diseño del modelo:

```
In [54]: y, X = patsy.dmatrices('v2 ~ key1 + 0', data)
```

```
In [55]: X
```

```
Out[55]:
```

```
DesignMatrix with shape (8, 2)
```

	key1[a]	key1[b]
1	0	0
1	0	0
0	1	1
0	1	1
1	0	0
0	1	1
1	0	0
0	1	1

Terms:

```
'key1' (columns 0:2)
```

Las columnas numéricas se pueden interpretar como categóricas con la función C:

```
In [56]: y, X = patsy.dmatrices('v2 ~ C(key2)', data)
```

```
In [57]: X
```

```
Out[57]:
```

```
DesignMatrix with shape (8, 2)
```

	Intercept	C(key2)[T.1]
1	0	0
1	0	1
1	0	0
1	0	1

1	0
1	1
1	0
1	0

Terms:

'Intercept' (column 0)
 'C(key2)' (column 1)

Cuando se utilizan varios términos categóricos en un modelo, las cosas pueden ser más complicadas, pues se pueden incluir términos de interacción de la forma key1:key2, que se pueden usar, por ejemplo, en modelos de análisis de varianza (ANOVA):

```
In [58]: data['key2'] = data['key2'].map({0: 'zero', 1: 'one'})
```

```
In [59]: data
Out[59]:
```

	key1	key2	v1	v2
0	a	zero	1	-1.0
1	a	one	2	0.0
2	b	zero	3	2.5
3	b	one	4	-0.5
4	a	zero	5	4.0
5	b	one	6	-1.2
6	a	zero	7	0.2
7	b	zero	8	-1.7

```
In [60]: y, X = patsy.dmatrices('v2 ~ key1 + key2', data)
```

```
In [61]: X
Out[61]:
```

DesignMatrix with shape (8, 3)

Intercept	key1[T.b]	key2[T.zero]
1	0	1

1	0	0
1	1	1
1	1	0
1	0	1
1	1	0
1	0	1
1	1	1

Terms:

- 'Intercept' (column 0)
- 'key1' (column 1)
- 'key2' (column 2)

```
In [62]: y, X = patsy.dmatrices('v2 ~ key1 + key2 + key1:key2', data)
```

```
In [63]: X
```

```
Out[63]:
```

DesignMatrix with shape (8, 4)

Intercept	key1[T.b]	key2[T.zero]	key1[T.b]:key2[T.zero]
1	0	1	0
1	0	0	0
1	1	1	1
1	1	0	0
1	0	1	0
1	1	0	0
1	0	1	0
1	1	1	1

Terms:

- 'Intercept' (column 0)
- 'key1' (column 1)
- 'key2' (column 2)
- 'key1:key2' (column 3)

Patsy ofrece otras formas de transformar datos categóricos, incluyendo transformaciones para términos con un determinado orden. La documentación en línea ofrece más información al respecto.

12.3 Introducción a statsmodels

La librería de Python statsmodels (<http://www.statsmodels.org>) se emplea para ajustar muchos tipos de modelos estadísticos, realizar pruebas estadísticas y explorar y visualizar datos. Esta librería contiene más métodos estadísticos frequentistas «clásicos», mientras que los bayesianos y de aprendizaje automático se encuentran en otras librerías.

Entre los tipos de modelos que se pueden encontrar en statsmodels se incluyen:

- Modelos lineales, lineales generalizados y lineales robustos.
- Modelos lineales de efectos mixtos.
- Métodos de análisis de varianza (ANOVA).
- Procesos de series temporales y modelos de espacio de estado.
- Método generalizado de momentos.

En las siguientes páginas usaremos varias herramientas básicas de statsmodels y exploraremos el modo en que se utilizan las interfaces de creación de modelos con fórmulas Patsy y objetos DataFrame de pandas. Si no estaba ya instalada statsmodels en la sección anterior sobre Patsy, ahora es el momento de hacerlo con:

```
conda install statsmodels
```

Estimación de modelos lineales

En statsmodels hay varios tipos de modelos de regresión lineal, desde el más básico (por ejemplo, de mínimos cuadrados ordinarios) al más complejo (por ejemplo, mínimos cuadrados iterativamente reponderados).

Los modelos lineales de statsmodels tienen dos interfaces principales distintas: basadas en arrays y basadas en fórmulas. A ellas se accede mediante estas importaciones del módulo API:

```
import statsmodels.api as sm  
import statsmodels.formula.api as smf
```

Para mostrar cómo se utilizan, generamos un modelo lineal a partir de unos cuantos datos aleatorios y ejecutamos el siguiente código en una celda Jupyter:

```
# Para que el ejemplo sea reproducible  
rng = np.random.default_rng(seed=12345)  
  
def dnorm(mean, variance, size=1):  
    if isinstance(size, int):  
        size = size,  
    return mean + np.sqrt(variance) *  
        rng.standard_normal(*size)  
  
N = 100  
X = np.c_[dnorm(0, 0.4, size=N),  
          dnorm(0, 0.6, size=N),  
          dnorm(0, 0.2, size=N)]  
  
eps = dnorm(0, 0.1, size=N)  
beta = [0.1, 0.3, 0.5]  
  
y = np.dot(X, beta) + eps
```

Aquí he escrito el modelo «real» con los parámetros beta conocidos. En este caso, `dnorm` es una función auxiliar para generar datos normalmente distribuidos con una media y varianza determinadas. Así que ahora tenemos:

```
In [66]: X[:5]  
Out[66]:
```

```
array([
       [-0.9005, -0.1894, -1.0279],
       [ 0.7993, -1.546 , -0.3274],
       [-0.5507, -0.1203,  0.3294],
       [-0.1639,  0.824 ,  0.2083],
       [-0.0477, -0.2131, -0.0482]])
```

```
In [67]: y[:5]
Out[67]: array([-0.5995, -0.5885,  0.1856, -0.0075, -0.0154])
```

Un modelo lineal se suele ajustar con un término de interceptación, como vimos antes con Patsy. La función `sm.add_constant` puede añadir una columna de interceptación a una matriz existente:

```
In [68]: X_model = sm.add_constant(X)

In [69]: X_model[:5]
Out[69]:
array([[ 1. , -0.9005, -0.1894, -1.0279],
       [ 1. ,  0.7993, -1.546 , -0.3274],
       [ 1. , -0.5507, -0.1203,  0.3294],
       [ 1. , -0.1639,  0.824 ,  0.2083],
       [ 1. , -0.0477, -0.2131, -0.0482]])
```

La clase `sm.OLS` puede ajustar una regresión de mínimos cuadrados ordinarios:

```
In [70]: model = sm.OLS(y, X)
```

El método `fit` del modelo devuelve un objeto con los resultados de la regresión que contiene parámetros estimados del modelo y otros diagnósticos:

```
In [71]: results = model.fit()

In [72]: results.params
Out[72]: array([0.0668,  0.268 ,  0.4505])
```

El método `summary` sobre `results` puede obtener en pantalla un modelo que detalla el resultado diagnóstico del modelo:

```
In [73]: print(results.summary())
OLS Regression Results
```

```
=====
=====
```

Dep.	y	R-squared
Variable:	(uncentered):	
0.469		
Model:	OLS	Adj. R-squared
		(uncentered):
0.452		
Method:	Least	F-statistic:
	Squares	
28.51		
Date:	Fri, 12 Aug	Prob (F-statistic):
	2022	2.66e-
		13
Time:	14:09:18	Log-Likelihood:
		-25.611
No.		AIC:
Observations:	100	57.22
Df Residuals:	97	BIC:
Df Model:	3	65.04
Covariance		
Type:	nonrobust	

```
=====
=====
```

```
=====
```

	coef	std err	t	P> t	[0.025	0.975]
x1	0.0668	0.054	1.243	0.217	-0.040	0.174
x2	0.2680	0.042	6.313	0.000	0.184	0.352
x3	0.4505	0.068	6.605	0.000	0.315	0.586

```
=====
=====
```

```
=====
```

Omnibus:	0.435	Durbin-Watson:	1.869
Prob(Omnibus):	0.805	Jarque-Bera (JB):	0.301

```
Skew:          0.134      Prob(JB):        0.860
Kurtosis:      2.995      Cond. No.:       1.64
```

```
=====
=====
```

Notes:

[1] R² is computed without centering (uncentered) since the model does not contain a constant.

[2] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Aquí, a los nombres de parámetros se les han dado los nombres genéricos x1, x2, etc. Supongamos en vez de esto que todos los parámetros del modelo están en un dataframe:

```
In [74]: data = pd.DataFrame(X, columns=['col0', 'col1', 'col2'])
```

```
In [75]: data['y'] = y
```

```
In [76]: data[:5]
```

```
Out[76]:
```

	col0	col1	col2	y
0	-0.900506	-0.189430	-1.027870	-0.599527
1	0.799252	-1.545984	-0.327397	-0.588454
2	-0.550655	-0.120254	0.329359	0.185634
3	-0.163916	0.824040	0.208275	-0.007477
4	-0.047651	-0.213147	-0.048244	-0.015374

Ahora podemos utilizar la API de la fórmula de statsmodels y las cadenas de texto de la fórmula Patsy:

```
In [77]: results = smf.ols('y ~ col0 + col1 + col2', data=data).fit()
```

```
In [78]: results.params
```

```
Out[78]:
```

```
Intercept -0.020799
```

```
col0 0.065813
```

```
col1 0.268970
col2 0.449419
dtype: float64

In [79]: results.tvalues
Out[79]:

Intercept -0.652501
col0 1.219768
col1 6.312369
col2 6.567428

dtype: float64
```

Podemos observar que statsmodels ha devuelto resultados como un objeto Series con los nombres de columna del dataframe añadidos. Tampoco es necesario que utilicemos add_constant al emplear fórmulas y objetos pandas.

Dados nuevos datos de fuera de la muestra, podemos calcular valores predichos utilizando los parámetros estimados del modelo:

```
In [80]: results.predict(data[:5])
Out[80]:
```

0	-0.592959
1	-0.531160
2	0.058636
3	0.283658
4	-0.102947

```
dtype: float64
```

En statsmodels hay muchas herramientas adicionales para análisis, diagnóstico y visualización de resultados de modelos lineales que se pueden explorar. También hay otros tipos de modelos lineales, aparte del de mínimos cuadrados ordinarios.

Estimación de procesos de series temporales

Otra clase de modelos de statsmodels es la que se emplea para analizar las series temporales. Entre ellos están los procesos autorregresivos, el filtrado de Kalman y otros modelos de espacio de estado, y los modelos autorregresivos multivariantes.

Simulemos algunos datos de series temporales con una estructura autorregresiva y ruido. Ejecutemos lo siguiente en Jupyter:

```
init_x = 4
values = [init_x, init_x]
N = 1000

b0 = 0.8
b1 = -0.4
noise = dnorm(0, 0.1, N)
for i in range(N):

    new_x = values[-1] * b0 + values[-2] * b1 + noise[i]
    values.append(new_x)
```

Estos datos tienen una estructura AR(2) (de dos retardos) con parámetros 0.8 y -0.4 . Al ajustar un modelo AR, es posible que no sepamos el número de términos retardados que hay que incluir, de modo que podemos ajustar el modelo con un número de retardos mayor:

```
In [82]: from statsmodels.tsa.ar_model import AutoReg
In [83]: MAXLAGS = 5
In [84]: model = AutoReg(values, MAXLAGS)
In [85]: results = model.fit()
```

Los parámetros estimados de los resultados tienen primero la interceptación, y después las estimaciones de los dos primeros retardos:

```
In [86]: results.params
Out[86]: array([ 0.0235,  0.8097, -0.4287, -0.0334,  0.0427,
 -0.0567])
```

Ofrecer una información más detallada de estos modelos y una explicación del modo en que se pueden interpretar sus resultados queda más allá del alcance de este libro, pero en la documentación de statsmodels hay mucho más por descubrir.

12.4 Introducción a scikit-learn

La librería scikit-learn (<http://scikit-learn.org>) es uno de los juegos de herramientas de aprendizaje automático de Python de propósito general más utilizados y fiables. Contiene una amplia selección de métodos estándares de aprendizaje automático supervisados y no supervisados, con herramientas para la selección y evaluación de modelos, la transformación y carga de datos y la persistencia de los modelos. Estos modelos pueden emplearse para tareas de clasificación, agrupamiento, predicción, y otras habituales. Se puede instalar scikit-learn desde conda de este modo:

```
conda install scikit-learn
```

Existen excelentes recursos en línea y en papel para adquirir conocimientos sobre el aprendizaje automático y sobre cómo aplicar librerías como scikit-learn para resolver problemas reales. En esta sección daré una breve muestra del estilo de la API de scikit-learn.

La integración de pandas en scikit-learn ha mejorado notablemente en los últimos años, y para cuando este libro esté en sus manos posiblemente haya mejorado todavía más. Animo a mis lectores a revisar la documentación más reciente del proyecto.

Como ejemplo para este capítulo voy a utilizar un conjunto de datos ya clásico de una competición Kaggle (<https://www.kaggle.com/c/titanic>) sobre las tasas de supervivencia de pasajeros del Titanic en 1912. Cargaremos los conjuntos de datos de entrenamiento y prueba utilizando pandas:

```
In [87]: train = pd.read_csv('datasets/titanic/train.csv')
```

```
In [88]: test = pd.read_csv('datasets/titanic/test.csv')
```

```
In [89]: train.head(4)
```

```
Out[89]:
```

	PassengerId	Survived	Pclass	\
0	1	0	3	
1	2	1	1	
2	3	1	3	
3	4	1	1	

	Name	Sex	Age	SibSp	\
0	Braund, Mr. Owen Harris	male	22.0	1	
1	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	female	38.0	1	
2	Heikkinen, Miss. Laina	female	26.0	0	
3	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	

	Parch	Ticket	Fare	Cabin	Embarked
0	0	A/5 21171	7.2500	NaN	S
1	0	PC 17599	71.2833	C85	C
2	0	STON/O2. 3101282	7.9250	NaN	S
3	0	113803	53.1000	C123	S

Generalmente, a las librerías como statsmodels y scikit-learn no se les pueden pasar datos ausentes, de modo que miramos las columnas para ver si hay algunas que contienen datos de este tipo:

```
In [90]: train.isna().sum()
```

```
Out[90]:
```

PassengerId	0
Survived	0
Pclass	0
Name	0
Sex	0
Age	177
SibSp	0

```
Parch          0
Ticket         0
Fare           0
Cabin          687
Embarked       2
```

dtype: int64

```
In [91]: test.isna().sum()
Out[91]:
```

```
PassengerId    0
Pclass          0
Name            0
Sex             0
Age             86
SibSp           0
Parch           0
Ticket          0
Fare            1
Cabin          327
Embarked        0
```

dtype: int64

En ejemplos de estadísticas y aprendizaje automático como este, una tarea muy normal es predecir si un pasajero sobreviviría basándose en las características de los datos. Un modelo se ajusta a un conjunto de datos de entrenamiento y después se evalúa según un conjunto de datos de prueba de fuera de la muestra.

Me gustaría utilizar Age como predictor, pero tiene datos ausentes. Hay varias maneras de hacer imputación de datos ausentes, pero lo haré de un modo sencillo y usaré la mediana del conjunto de datos de entrenamiento para llenar los valores nulos de ambas tablas:

```
In [92]: impute_value = train['Age'].median()
```

```
In [93]: train['Age'] = train['Age'].fillna(impute_value)
```

```
In [94]: test['Age'] = test['Age'].fillna(impute_value)
```

Ahora tenemos que especificar nuestros modelos. Añado una columna IsFemale como versión codificada de la columna ‘Sex’:

```
In [95]: train['IsFemale'] = (train['Sex'] == 'female').astype(int)
```

```
In [96]: test['IsFemale'] = (test['Sex'] == 'female').astype(int)
```

A continuación decidimos algunas variables del modelo y creamos arrays NumPy:

```
In [97]: predictors = ['Pclass', 'IsFemale', 'Age']
```

```
In [98]: X_train = train[predictors].to_numpy()
```

```
In [99]: X_test = test[predictors].to_numpy()
```

```
In [100]: y_train = train['Survived'].to_numpy()
```

```
In [101]: X_train[:5]
```

```
Out[101]:
```

```
array([
       [ 3.,  0., 22.],
       [ 1.,  1., 38.],
       [ 3.,  1., 26.],
       [ 1.,  1., 35.],
       [ 3.,  0., 35.]])
```

```
In [102]: y_train[:5]
```

```
Out[102]: array([0, 1, 1, 1, 0])
```

Mi intención no es afirmar que este es un buen modelo o que estas características están correctamente diseñadas. Utilizamos el modelo LogisticRegression de scikit-learn y creamos una instancia del modelo:

```
In [103]: from sklearn.linear_model import LogisticRegression
```

```
In [104]: model = LogisticRegression()
```

Podemos ajustar este modelo a los datos de entrenamiento empleando su método `fit`:

```
In [105]: model.fit(X_train, y_train)
Out[105]: LogisticRegression()
```

Ahora podemos formar predicciones para el conjunto de datos de prueba utilizando `model.predict`:

```
In [106]: y_predict = model.predict(X_test)

In [107]: y_predict[:10]
Out[107]: array([0, 0, 0, 0, 1, 0, 1, 0, 1, 0])
```

Si tuviéramos los valores reales del conjunto de datos de prueba, podríamos calcular un porcentaje de precisión o alguna otra métrica de error:

```
(y_true == y_predict).mean()
```

En la práctica, a menudo hay muchas capas adicionales de complejidad en el entrenamiento de modelos. Muchos modelos tienen parámetros que se pueden afinar, y existen técnicas, como la validación cruzada, que se pueden emplear para evitar el sobreajuste de los datos de entrenamiento y obtener con frecuencia un mejor rendimiento predictivo o una mayor robustez en los datos nuevos.

La validación cruzada funciona dividiendo los datos de entrenamiento para simular una predicción de fuera de la muestra. Basándonos en un marcador de precisión del modelo, como el error medio cuadrático, se puede realizar una búsqueda en rejilla de los parámetros del modelo. Algunos modelos, como la regresión logística, tienen clases de estimadores con validación cruzada integrada. Por ejemplo, la clase `LogisticRegressionCV` se puede utilizar con un parámetro que indique cuál debe ser el grado de precisión de una búsqueda en rejilla para realizarla sobre el parámetro `c` de regularización del modelo:

```
In [108]: from sklearn.linear_model import LogisticRegressionCV
```

```
In [109]: model_cv = LogisticRegressionCV(Cs=10)
```

```
In [110]: model_cv.fit(X_train, y_train)
Out[110]: LogisticRegressionCV()
```

Para realizar validación cruzada a mano, podemos usar la función auxiliar `cross_val_score`, que se encarga del proceso de división de los datos. Por ejemplo, para hacer la validación cruzada de nuestro modelo con cuatro divisiones no superpuestas de los datos de entrenamiento, podemos hacer lo siguiente:

```
In [111]: from sklearn.model_selection import cross_val_score
```

```
In [112]: model = LogisticRegression(C=10)
```

```
In [113]: scores = cross_val_score(model, X_train, y_train, cv=4)
```

```
In [114]: scores
```

```
Out[114]: array([0.7758, 0.7982, 0.7758, 0.7883])
```

La métrica de marcadores predeterminada depende del modelo, pero es posible elegir una función de marcador explícita. Los modelos de validación cruzada tardan más en entrenarse, pero a menudo ofrecen un rendimiento del modelo mucho mejor.

12.5 Conclusión

Aunque solo hemos añadido la superficie de algunas de las librerías para creación de modelos de Python existentes, cada vez hay más marcos de referencia para distintos tipos de estadísticas y aprendizaje automático, implementados directamente en Python o con una interfaz de usuario de Python.

Este libro se centra especialmente en la manipulación de datos, pero hay otros muchos dedicados a los modelos y a herramientas de ciencia de datos. Algunos muy buenos son los siguientes:

- *Introduction to Machine Learning with Python*, de Andreas Müller y Sarah Guido (O'Reilly).
- *Python Data Science Handbook*, de Jake VanderPlas (O'Reilly).
- *Ciencia de datos desde cero: Principios básicos con Python*, de Joel Grus (Anaya Multimedia).
- *Python Machine Learning*, de Sebastian Raschka y Vahid Mirjalili (Packt Publishing).
- *Aprende Machine Learning con Scikit-Learn, Keras y TensorFlow*, de Aurélien Géron (Anaya Multimedia).

Aunque los libros pueden ser unos recursos de gran valor para aprender, con frecuencia se quedan obsoletos cuando el software de código abierto subyacente cambia. Es una buena idea estar familiarizados con la documentación de los distintos marcos de referencia estadísticos o de aprendizaje automático para mantenerse actualizados sobre las características y las API más recientes.

Ejemplos de análisis de datos

Ahora que hemos llegado al capítulo final de este libro, echaremos un vistazo a algunos conjuntos de datos reales. Para cada uno de ellos usaremos las técnicas presentadas desde el comienzo para extraer significado de los datos sin procesar. Las técnicas mostradas se pueden aplicar a cualquier tipo de datos. Este capítulo contiene una colección de conjuntos de datos de ejemplo que se pueden utilizar para practicar con las herramientas de este libro.

Los conjuntos de datos de ejemplo se pueden encontrar en el repositorio GitHub que acompaña al libro (<http://github.com/wesm/pydata-book>). Si no es posible acceder a GitHub, también se pueden conseguir en el repositorio duplicado de Gitee (<https://gitee.com/wesmckinn/pydata-book>).

13.1 Datos Bitly de 1.USA.gov

En 2011, el servicio de acortamiento de URL Bitly (<https://bitly.com>) se asoció con el sitio web del gobierno de los Estados Unidos USA.gov (<https://www.usa.gov>) para ofrecer una fuente de datos anónimos recogidos de usuarios que acortan enlaces terminados en .gov or .mil. En 2011 ofrecían datos en línea (además de una captura cada hora) como archivos de texto descargables. Este servicio está cerrado en el momento de escribir esto (2022), pero hemos logrado conservar uno de los archivos de datos para los ejemplos de libro.

En el caso de las capturas, cada línea de cada archivo contiene una forma común de datos web denominada JSON, que significa *JavaScript Object Notation* (notación de objetos JavaScript). Por ejemplo, si leemos solamente la primera línea de un archivo, podemos ver algo así:

```
In [5]: path = "datasets/bitly_usagov/example.txt"  
In [6]: with open(path) as f:  
....:     print(f.readline())  
....:  
  
{ "a": "Mozilla\\5.0 (Windows NT 6.1; WOW64) AppleWebKit\\535.11  
(KHTML, like Gecko) Chrome\\17.0.963.78 Safari\\535.11", "c": "US", "nk":  
1,  
"tz": "America\\New_York", "gr": "MA", "g": "A6q0VH", "h": "wfLQtf", "l":  
"orofrog", "al": "en-US,en;q=0.8", "hh": "1.usa.gov", "r":  
"http:\\\\www.facebook.com\\\\1\\\\7AQEFzjsi\\\\1.usa.gov\\\\wfLQtf", "u":  
"http:\\\\www.ncbi.nlm.nih.gov\\\\pubmed\\\\22415991", "t": 1331923247,  
"hc":
```

```
1331822918, "cy": "Danvers", "ll": [ 42.576698, -70.954903 ] }
```

Python tiene librerías internas y externas para convertir una cadena de texto JSON en un diccionario Python. Aquí utilizaremos el módulo json y su función loads invocada en cada línea en el archivo de muestra que hemos descargado:

```
import json
with open(path) as f:
    records = [json.loads(line) for line in f]
```

El objeto resultante records es ahora una lista de diccionarios Python:

```
In [18]: records[0]
Out[18]:
{'a': 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like Gecko)
Chrome/17.0.963.78 Safari/535.11',
 'al': 'en-US,en;q=0.8',
 'c': 'US',
 'cy': 'Danvers',
 'g': 'A6qOVH',
 'gr': 'MA',
 'h': 'wfLQtf',
 'hc': 1331822918,
 'hh': '1.usa.gov',
 'l': 'orofrog',
 'll': [42.576698, -70.954903],
 'nk': 1,
 'r': 'http://www.facebook.com/l/7AQEFzjSi/1.usa.gov/wfLQtf',
 't': 1331923247,
 'tz': 'America/New_York',
 'u': 'http://www.ncbi.nlm.nih.gov/pubmed/22415991'}
```

Recuento de zonas horarias en Python puro

Supongamos que estuviéramos interesados en averiguar qué zonas horarias ocurren con más frecuencia en el conjunto de datos (el campo tz). Hay muchas formas de hacer esto. Primero, extraemos de nuevo una lista de zonas horarias empleando una comprensión de lista:

```
In [15]: time_zones = [rec["tz"] for rec in records]
```

```
KeyError          Traceback (most recent call last)

<ipython-input-15-abdeba901c13> in <module>
      1 time_zones = [rec["tz"] for rec in records]
<ipython-input-15-abdeba901c13> in <listcomp>(.0)
```

```
—> 1 time_zones = [rec["tz"] for rec in records]
KeyError: 'tz'
```

¡Vaya! Resulta que no todos los registros tienen un campo de zona horaria. Podemos gestionar esto añadiendo la comprobación `if "tz" in rec` al final de la comprensión de lista:

```
In [16]: time_zones = [rec["tz"] for rec in records if "tz" in rec]
```

```
In [17]: time_zones[:10]
Out[17]:
['America/New_York',
 'America/Denver',
 'America/New_York',
 'America/Sao_Paulo',
 'America/New_York',
 'America/New_York',
 'Europe/Warsaw',
 '',
 '',
 '']
```

Basta con mirar las primeras diez zonas horarias para comprobar que algunas son desconocidas (cadena de texto vacía). También se pueden filtrar, pero por el momento las dejaremos. Después, para producir recuentos por zona horaria, mostraré dos métodos: uno más difícil (utilizando solo la librería estándar de Python) y otro más sencillo (empleando pandas). Una forma de hacer el recuento es usar un diccionario que almacene recuentos mientras iteramos por las zonas horarias:

```
def get_counts(sequence):
    counts = {}
    for x in sequence:
        if x in counts:
            counts[x] += 1
        else:
            counts[x] = 1
    return counts
```

Empleando herramientas más avanzadas de la librería estándar de Python, se puede escribir lo mismo de una forma más resumida:

```
from collections import defaultdict
def get_counts2(sequence):

    counts = defaultdict(int) # los valores se inicializan a 0
    for x in sequence:
        counts[x] += 1
    return counts
```

Pongo esta lógica en una función simplemente para poder reutilizarla. Para usarla con las zonas horarias, basta con pasar la lista `time_zones`:

```
In [20]: counts = get_counts(time_zones)
In [21]: counts["America/New_York"]
Out[21]: 1251
In [22]: len(time_zones)
Out[22]: 3440
```

Si queremos las diez zonas horarias principales y sus recuentos, podemos hacer una lista de tuplas por `(count, timezone)` y ordenarla:

```
def top_counts(count_dict, n=10):
    value_key_pairs = [(count, tz) for tz, count in count_dict.items()]
    value_key_pairs.sort()
    return value_key_pairs[-n:]
```

Entonces tenemos:

```
In [24]: top_counts(counts)
Out[24]:
[(33, 'America/Sao_Paulo'),
(35, 'Europe/Madrid'),
(36, 'Pacific/Honolulu'),
(37, 'Asia/Tokyo'),
(74, 'Europe/London'),
(191, 'America/Denver'),
(382, 'America/Los_Angeles'),
(400, 'America/Chicago'),
(521, ''),
(1251, 'America/New_York')]
```

Si buscáramos en la librería estándar de Python, encontraríamos la clase `collections.Counter`, que simplifica todavía más esta tarea:

```
In [25]: from collections import Counter
In [26]: counts = Counter(time_zones)
In [27]: counts.most_common(10)
Out[27]:
[('America/New_York', 1251),
('', 521),
('America/Chicago', 400),
('America/Los_Angeles', 382),
('America/Denver', 191),
('Europe/London', 74),
('Asia/Tokyo', 37),
```

```
('Pacific/Honolulu', 36),  
('Europe/Madrid', 35),  
('America/Sao_Paulo', 33)]
```

Recuento de zonas horarias con pandas

Es posible crear un objeto DataFrame a partir del conjunto original de registros pasándole a pandas.DataFrame la lista de registros:

```
In [28]: frame = pd.DataFrame(records)
```

Podemos consultar cierta información básica sobre este nuevo dataframe, como los nombres de las columnas, o el número de valores ausentes, utilizando frame.info():

```
In [29]: frame.info()  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 3560 entries, 0 to 3559  
Data columns (total 18 columns):
```

#	Column	Non-Null Count	Dtype
0	a	3440 non-null	object
1	c	2919 non-null	object
2	nk	3440 non-null	float64
3	tz	3440 non-null	object
4	gr	2919 non-null	object
5	g	3440 non-null	object
6	h	3440 non-null	object
7	l	3440 non-null	object
8	al	3094 non-null	object
9	hh	3440 non-null	object
10	r	3440 non-null	object
11	u	3440 non-null	object
12	t	3440 non-null	float64
13	hc	3440 non-null	float64
14	cy	2919 non-null	object
15	ll	2919 non-null	object
16	_heartbeat_	120 non-null	float64
17	kw	93 non-null	object

```
dtypes: float64(4), object(14)  
memory usage: 500.8+ KB
```

```
In [30]: frame["tz"].head()  
Out[30]:
```

0	America/New_York
1	America/Denver
2	America/New_York

```
3
```

```
4
```

```
America/Sao_Paulo
```

```
America/New_York
```

```
Name: tz, dtype: object
```

El resultado mostrado para `frame` es la vista de resumen, mostrada para grandes objetos `DataFrame`. Podemos utilizar después el método `value_counts` para el objeto `Series`:

```
In [31]: tz_counts = frame["tz"].value_counts()
```

```
In [32]: tz_counts.head()
```

```
Out[32]:
```

America/New_York	1251
	521
America/Chicago	400
America/Los_Angeles	382
America/Denver	191

```
Name: tz, dtype: int64
```

Estos datos se pueden visualizar usando `matplotlib`, y los gráficos pueden ser un poco más bonitos si rellenamos un valor sustituto para los datos de zona horaria desconocidos o ausentes que aparezcan en los registros. Reemplazamos los valores ausentes con el método `fillna` y empleamos indexación de array booleano para las cadenas de texto vacías:

```
In [33]: clean_tz = frame["tz"].fillna("Missing")
```

```
In [34]: clean_tz[clean_tz == ""] = "Unknown"
```

```
In [35]: tz_counts = clean_tz.value_counts()
```

```
In [36]: tz_counts.head()
```

```
Out[36]:
```

America/New_York	1251
Unknown	521
America/Chicago	400
America/Los_Angeles	382
America/Denver	191

```
Name: tz, dtype: int64
```

En este momento podemos usar el paquete `seaborn` (<http://seaborn.pydata.org>) para crear un gráfico de barras horizontales (la figura 13.1 muestra la visualización resultante):

```
In [38]: import seaborn as sns
```

```
In [39]: subset = tz_counts.head()
```

```
In [40]: sns.barplot(y=subset.index, x=subset.to_numpy())
```

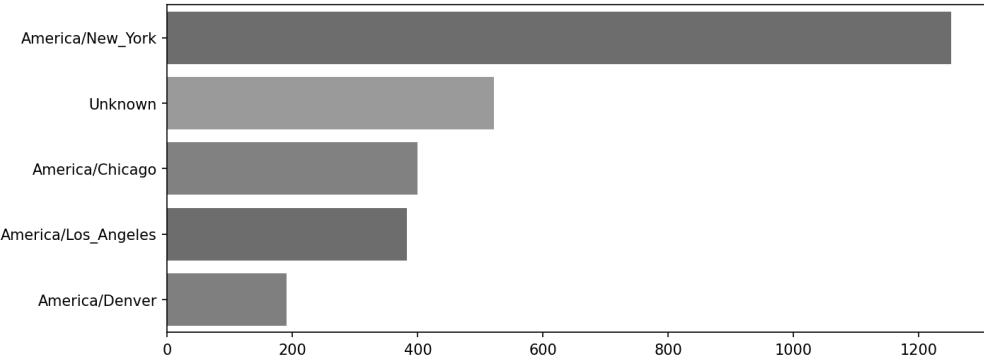


Figura 13.1. Principales zonas horarias de los datos de muestra de 1.usa.gov.

El campo `a` contiene información sobre el navegador, el dispositivo o la aplicación empleada para realizar el acortamiento de URL:

```
In [41]: frame["a"][1]
Out[41]: 'GoogleMaps/RochesterNY'

In [42]: frame["a"][50]
Out[42]: 'Mozilla/5.0 (Windows NT 5.1; rv:10.0.2) Gecko/20100101
Firefox/10.0.2'

In [43]: frame["a"][51][:50] # línea larga
Out[43]: 'Mozilla/5.0 (Linux; U; Android 2.2.2; en-us; LG-P9'
```

Analizar toda la información interesante de estas cadenas de texto «agentes» puede parecer una tarea desalentadora. Una posible estrategia es separar la primera parte de la cadena de texto (que corresponde más o menos al navegador empleado) y hacer otro resumen del comportamiento del usuario:

```
In [44]: results = pd.Series([x.split()[0] for x in frame["a"].dropna()])
In [45]: results.head(5)
Out[45]:
0                               Mozilla/5.0
1           GoogleMaps/RochesterNY
2                               Mozilla/4.0
3                               Mozilla/5.0
4                               Mozilla/5.0

dtype: object

In [46]: results.value_counts().head(8)
Out[46]:
Mozilla/5.0                           2594
Mozilla/4.0                            601
GoogleMaps/RochesterNY                  121
```

Opera/9.80	34
TEST_INTERNET_AGENT_2	4
GoogleProducer	21
Mozilla/6.0	5
BlackBerry8520/5.0.0.681	4
dtype: int64	

Ahora supongamos que queremos descomponer las zonas horarias principales en usuarios de Windows y de otros sistemas operativos. Para simplificar, digamos que un usuario está en Windows si la cadena de texto “Windows” aparece en la cadena agente. Como faltan algunos de los agentes, los excluiremos de los datos:

```
In [47]: cframe = frame[frame["a"].notna()].copy()
```

Queremos entonces calcular un valor para saber si cada fila es o no Windows:

```
In [48]: cframe["os"] = np.where(cframe["a"].str.contains("Windows"),
```

```
....:           "Windows", "Not Windows")
```

```
In [49]: cframe["os"].head(5)
Out[49]:
```

0	Windows
1	Not Windows
2	Windows
3	Not Windows
4	Windows

```
Name: os, dtype: object
```

A continuación, podemos agrupar los datos por su columna de zona horaria y por esta nueva lista de sistemas operativos:

```
In [50]: by_tz_os = cframe.groupby(["tz", "os"])
```

Los recuentos de grupo, análogos a la función value_counts, se pueden calcular con size. Este resultado se reconfigura después como una tabla con unstack:

```
In [51]: agg_counts = by_tz_os.size().unstack().fillna(0)
```

```
In [52]: agg_counts.head()
Out[52]:
```

os	Not Windows	Windows
tz	245.0	276.0
Africa/Cairo	0.0	3.0
Africa/Casablanca	0.0	1.0

Africa/Ceuta	0.0	2.0
Africa/Johannesburg	0.0	1.0

Por último, seleccionemos las principales zonas horarias a nivel global. Para ello, construyo un array de índice indirecto a partir de los recuentos de filas en agg_counts. Tras calcular los recuentos de filas con agg_counts.sum("columns"), puedo llamar a argsort() para obtener un array de índice que se pueda utilizar para ordenar de forma ascendente:

```
In [53]: indexer = agg_counts.sum("columns").argsort()
```

```
In [54]: indexer.values[:10]
Out[54]: array([24, 20, 21, 92, 87, 53, 54, 57, 26, 55])
```

Yo utilizo take para seleccionar las filas en ese orden, y después separar las últimas diez filas (los valores más grandes):

```
In [55]: count_subset = agg_counts.take(indexer[-10:])
```

```
In [56]: count_subset
Out[56]:
```

os	NotWindows	Windows
tz		
America/Sao_Paulo	13.0	20.0
Europe/Madrid	16.0	19.0
Pacific/Honolulu	0.0	36.0
Asia/Tokyo	2.0	35.0
Europe/London	43.0	31.0
America/Denver	132.0	59.0
America/Los_Angeles	130.0	252.0
America/Chicago	115.0	285.0
245.0	276.0	
America/New_York	339.0	912.0

pandas tiene un oportuno método llamado nlargest que hace lo mismo:

```
In [57]: agg_counts.sum(axis="columns").nlargest(10)
Out[57]:
```

tz		
America/New_York	1251.0	
	521.0	
America/Chicago	400.0	
America/Los_Angeles	382.0	
America/Denver	191.0	
Europe/London	74.0	
Asia/Tokyo	37.0	

```
Pacific/Honolulu          36.0
Europe/Madrid              35.0
America/Sao_Paulo           33.0
```

```
dtype: float64
```

Después, esto se puede visualizar en un gráfico de barras agrupado que compara el número de usuarios Windows y no Windows, utilizando la función barplot de seaborn (figura 13.2). Primero llamo a count_subset.stack() y reinicio el índice para reordenar los datos y disponer así de una mejor compatibilidad con seaborn:

```
In [59]: count_subset = count_subset.stack()
In [60]: count_subset.name = "total"
In [61]: count_subset = count_subset.reset_index()
In [62]: count_subset.head(10)
Out[62]:
```

	tz	os	total
0	America/Sao_Paulo	Not Windows	13.0
1	America/Sao_Paulo	Windows	20.0
2	Europe/Madrid	Not Windows	16.0
3	Europe/Madrid	Windows	19.0
4	Pacific/Honolulu	Not Windows	0.0
5	Pacific/Honolulu	Windows	36.0
6	Asia/Tokyo	Not Windows	2.0
7	Asia/Tokyo	Windows	35.0
8	Europe/London	Not Windows	43.0
9	Europe/London	Windows	31.0

```
In [63]: sns.barplot(x="total", y="tz", hue="os", data=count_subset)
```

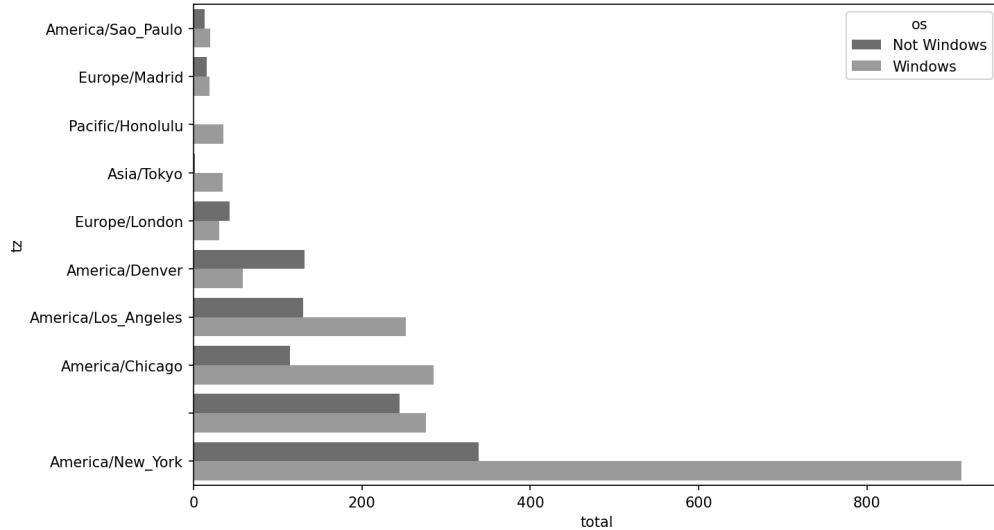


Figura 13.2. Principales zonas horarias por usuarios Windows y no Windows.

Es un poco difícil ver el porcentaje relativo de usuarios Windows en los grupos más pequeños, así que normalizaremos los porcentajes de grupo para que sumen 1:

```
def norm_total(group):

    group["normed_total"] = group["total"] / group["total"].sum()
    return group

results = count_subset.groupby("tz").apply(norm_total)
```

Después visualizamos esto en la figura 13.3:

```
In [66]: sns.barplot(x="normed_total", y="tz", hue="os", data=results)
```

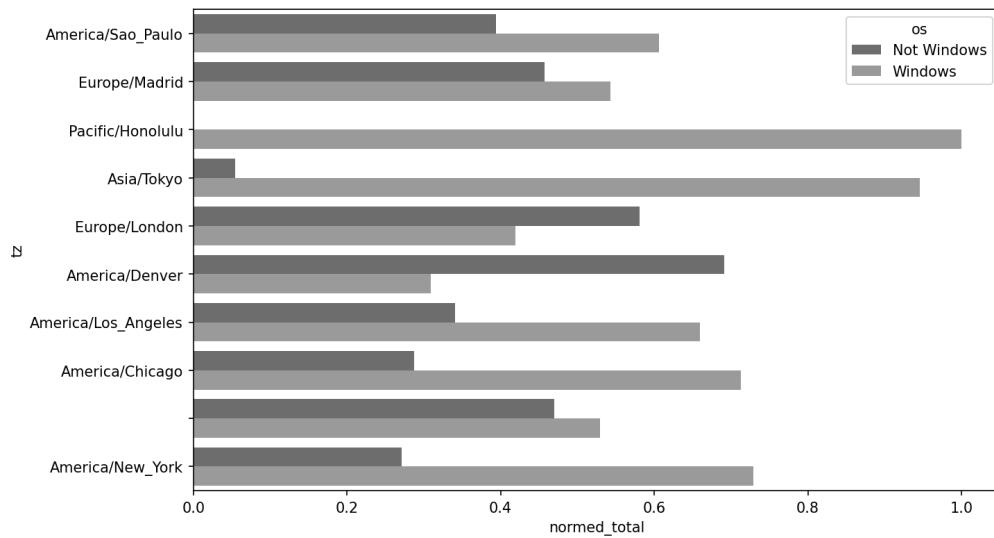


Figura 13.3. Porcentaje de usuarios Windows y no Windows en las principales zonas horarias.

Podríamos haber calculado la suma normalizada de una forma más eficiente utilizando el método `transform` con `groupby`:

```
In [67]: g = count_subset.groupby("tz")
In [68]: results2 = count_subset["total"] / g["total"].transform("sum")
```

13.2 Conjunto de datos MovieLens 1M

GroupLens Research (<https://grouplens.org/datasets/movielens>) ofrece distintas colecciones de datos de valoraciones de películas recogidos de usuarios de MovieLens a finales de los años 90 y principios de la década de 2000. Los datos proporcionan valoraciones de películas, metadatos de las mismas (géneros y año), así como datos demográficos sobre los usuarios (edad, código postal, sexo y ocupación). Estos datos suelen ser de interés en el desarrollo de sistemas de recomendación basados en algoritmos de aprendizaje automático. Aunque no vayamos a explorar este tipo de técnicas con detalle en este libro, sí voy a mostrar cómo segmentar conjuntos de datos como estos en la forma exacta que se necesite en cada caso.

El conjunto de datos MovieLens 1M contiene un millón de valoraciones recogidas de 6000 usuarios sobre 4000 películas. Se extiende a lo largo de tres tablas: valoraciones, información del usuario e información de la película. Podemos cargar cada una de las tablas en un objeto DataFrame de pandas utilizando `pandas.read_table`. Ejecutemos lo siguiente en una celda Jupyter:

```
unames = ["user_id", "gender", "age", "occupation", "zip"]
users = pd.read_table("datasets/movielens/users.dat", sep="::",
                      header=None, names=unames, engine="python")

rnames = ["user_id", "movie_id", "rating", "timestamp"]
ratings = pd.read_table("datasets/movielens/ratings.dat", sep="::",
                        header=None, names=rnames, engine="python")

mnames = ["movie_id", "title", "genres"]
movies = pd.read_table("datasets/movielens/movies.dat", sep="::",
                      header=None, names=mnames, engine="python")
```

Podemos verificar que todo fue bien revisando cada dataframe:

```
In [70]: users.head(5)
Out[70]:
```

	user_id	gender	age	occupation	zip
0	1	F	1	10	48067
1	2	M	56	16	70072
2	3	M	25	15	55117
3	4	M	45	7	02460
4	5	M	25	20	55455

In [71]: ratings.head(5)
Out[71]:

	user_id	movie_id	rating	timestamp
0	1	1193	5	978300760
1	1	661	3	978302109
2	1	914	3	978301968
3	1	3408	4	978300275
4	1	2355	5	978824291

In [72]: movies.head(5)
Out[72]:

	movie_id	title	genres
0	1	Toy Story (1995)	Animation Children's Comedy
1	2	Jumanji (1995)	Adventure Children's Fantasy
2	3	Grumpier Old Men (1995)	Comedy Romance
3	4	Waiting to Exhale (1995)	Comedy Drama
4	5	Father of the Bride Part II (1995)	Comedy

In [73]: ratings
Out[73]:

	user_id	movie_id	rating	timestamp
0	1	1193	5	978300760
1	1	661	3	978302109
2	1	914	3	978301968
3	1	3408	4	978300275
4	1	2355	5	978824291
...
1000204	6040	1091	1	956716541
1000205	6040	1094	5	956704887
1000206	6040	562	5	956704746
1000207	6040	1096	4	956715648
1000208	6040	1097	4	956715569

[1000209 rows x 4 columns]

Conviene tener en cuenta que las edades y las profesiones están codificadas como enteros, que indican grupos descritos en el archivo `README` del conjunto de datos. Analizar la distribución de los datos a lo largo de tres tablas no es una tarea sencilla; por ejemplo, supongamos que queremos calcular las valoraciones medias para una determinada película por sexo y profesión. Como veremos, es más cómodo hacer esto con todos los datos combinados en una sola tabla. Utilizando la función `merge` de pandas, combinamos primero `ratings` con `users`, y después ese resultado lo mezclamos con los datos de `movies`. El mismo pandas deduce qué columnas usar como claves de combinación basándose en los nombres superpuestos:

```
In [74]: data = pd.merge(pd.merge(ratings, users), movies)
```

```
In [75]: data  
Out[75]:
```

	user_id	movie_id	rating	timestamp	gender	age	occupation	zip
0	1	1193	5	978300760	F	1	10	48067
1	2	1193	5	978298413	M	56	16	70072
2	12	1193	4	978220179	M	25	12	32793
3	15	1193	4	978199279	M	25	7	22903
4	17	1193	5	978158471	M	50	1	95350
...
1000204	5949	2198	5	958846401	M	18	17	47901
1000205	5675	2703	3	976029116	M	35	14	30030
1000206	5780	2845	1	958153068	M	18	17	92886
1000207	5851	3607	5	957756608	F	18	20	55410
1000208	5938	2909	4	957273353	M	25	1	35401
	title	genres						
0	One Flew Over the Cuckoo's Nest (1975)	Drama						
1	One Flew Over the Cuckoo's Nest (1975)	Drama						
2	One Flew Over the Cuckoo's Nest (1975)	Drama						
3	One Flew Over the Cuckoo's Nest (1975)	Drama						
4	One Flew Over the Cuckoo's Nest (1975)	Drama						

```

1000204 Modulations Documentary
          (1998)
1000205   Broken
          Vessels Drama
          (1998)
1000206 White Boys Drama
          (1999)
1000207 One Little Indian Drama|Western
          (1973)
1000208 Five Wives,
          Three
          Secretaries Documentary
          and Me
          (1998)

```

[1000209 rows x 10 columns]

In [76]: data.iloc[0]

Out[76]:

user_id		1
movie_id		1193
rating		5
timestamp		978300760
gender		F
age		1
occupation		10
zip		48067
title	One Flew Over the Cuckoo's Nest	(1975)
genres		Drama

Name: 0, dtype: object

Para obtener las valoraciones medias para cada película agrupada por género, podemos usar el método pivot_table:

In [77]: mean_ratings = data.pivot_table("rating", index="title",

.....: columns="gender", aggfunc="mean")

In [78]: mean_ratings.head(5)

Out[78]:

	F	M
gender		
title		
\$1,000,000 Duck (1971)	3.375000	2.761905
'Night Mother (1986)	3.388889	3.352941
'Til There Was You (1997)	2.675676	2.733333

'burbs, The (1989)	2.793478	2.962085
...And Justice for All (1979)	3.828571	3.689024

Este código produjo otro dataframe, que contiene las valoraciones medias con los títulos de las películas como etiquetas de filas (el «índice») y el sexo del usuario como etiquetas de columnas. Primero filtro por las películas que recibieron al menos 250 valoraciones (un número arbitrario); para ello, agrupo los datos por título y utilizo `size()` para obtener una serie de tamaños de grupo para cada título:

```
In [79]: ratings_by_title = data.groupby("title").size()

In [80]: ratings_by_title.head()
Out[80]:
title

$1,000,000 Duck (1971)                                37
'Night Mother (1986)                                    70
'Til There Was You (1997)                               52
'burbs, The (1989)                                     303
...And Justice for All (1979)                            199

dtype: int64

In [81]: active_titles = ratings_by_title.index[ratings_by_title >= 250]

In [82]: active_titles
Out[82]:
Index([''burbs, The (1989)', '10 Things I Hate About You (1999)',
       '101 Dalmatians (1961)', '101 Dalmatians (1996)', '12 Angry Men (1957)',
       '13th Warrior, The (1999)', '2 Days in the Valley (1996)',
       '20,000 Leagues Under the Sea (1954)', '2001: A Space Odyssey (1968)',
       '2010 (1984)',
       ...,
       'X-Men (2000)', 'Year of Living Dangerously (1982)',
       'Yellow Submarine (1968)', 'You've Got Mail (1998)',
       'Young Frankenstein (1974)', 'Young Guns (1988)',
       'Young Guns II (1990)', 'Young Sherlock Holmes (1985)',
       ',Zero Effect (1998)', ',eXistenZ (1999)'],
      dtype='object', name='title', length=1216)
```

El índice de títulos que han recibido al menos 250 valoraciones se puede utilizar entonces para seleccionar las filas de `mean_ratings` empleando `.loc`:

```
In [83]: mean_ratings = mean_ratings.loc[active_titles]

In [84]: mean_ratings
Out[84]:
```

	F	M
gender		
title		
'burbs, The (1989)	2.793478	2.962085
10 Things I Hate About You (1999)	3.646552	3.311966
101 Dalmatians (1961)	3.791444	3.500000
101 Dalmatians (1996)	3.240000	2.911215
12 Angry Men (1957)	4.184397	4.328421
...
Young Guns (1988)	3.371795	3.425620
Young Guns II (1990)	2.934783	2.904025
Young Sherlock Holmes (1985)	3.514706	3.363344
Zero Effect (1998)	3.864407	3.723140
eXistenZ (1999)	3.098592	3.289086

[1216 rows x 2 columns]

Para ver las películas mejor calificadas por espectadoras femeninas, podemos ordenar por la columna F en orden descendente:

```
In [86]: top_female_ratings = mean_ratings.sort_values("F", ascending=False)
```

```
In [87]: top_female_ratings.head()
Out[87]:
```

	F	M
gender		
title		
Close Shave, A (1995)	4.644444	4.473795
Wrong Trousers, The (1993)	4.588235	4.478261
Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)	4.572650	4.464589
Wallace & Gromit: The Best of Aardman Animation (1996)	4.563107	4.385075
Schindler's List (1993)	4.562602	4.491415

Medición del desacuerdo en las valoraciones

Supongamos que queremos averiguar cuáles son las películas que más dividen a los espectadores masculinos y femeninos. Una forma de hacerlo es añadir una columna a mean_ratings que contenga la diferencia de medias y después ordenar por ella:

```
In [88]: mean_ratings["diff"] = mean_ratings["M"]-mean_ratings["F"]
```

Ordenar por "diff" produce las películas que tienen la mayor diferencia de valoración, de forma que podemos ver cuáles son las preferidas por las mujeres:

```
In [89]: sorted_by_diff = mean_ratings.sort_values("diff")
```

```
In [90]: sorted_by_diff.head()
Out[90]:
```

	F	M	diff
gender			
title			
Dirty Dancing (1987)	3.790378	2.959596	-0.830782
Jumpin' Jack Flash (1986)	3.254717	2.578358	-0.676359
Grease (1978)	3.975265	3.367041	-0.608224
Little Women (1994)	3.870588	3.321739	-0.548849
Steel Magnolias (1989)	3.901734	3.365957	-0.535777

Invertiendo el orden de las filas y sacando de nuevo las diez primeras, obtenemos las películas preferidas por los hombres, que las mujeres no valoraron tanto como ellos:

```
In [91]: sorted_by_diff[::-1].head()
Out[91]:
```

	F	M	diff
gender			
title			
Good, The Bad and The Ugly, The (1966)	3.494949	4.221300	0.726351
Kentucky Fried Movie, The (1977)	2.878788	3.555147	0.676359
Dumb & Dumber (1994)	2.697987	3.336595	0.638608
Longest Day, The (1962)	3.411765	4.031447	0.619682
Cable Guy, The (1996)	2.250000	2.863787	0.613787

Supongamos que en realidad queríamos las películas que más desacuerdo suscitaron entre los espectadores, independientemente de su sexo. El desacuerdo se puede medir por la varianza o la desviación estándar de las valoraciones. Para obtenerlo, primero calculamos la desviación estándar de la valoración por título y después filtramos por los títulos activos:

```
In [92]: rating_std_by_title = data.groupby("title")["rating"].std()
```

```
In [93]: rating_std_by_title = rating_std_by_title.loc[active_titles]
```

```
In [94]: rating_std_by_title.head()
```

```
Out[94]:
```

title		
'burbs, The (1989)		1.107760
10 Things I Hate About You (1999)		0.989815
101 Dalmatians (1961)		0.982103
101 Dalmatians (1996)		1.098717
12 Angry Men (1957)		0.812731

```
Name: rating, dtype: float64
```

Después, ordenamos de forma descendente y seleccionamos las primeras diez filas, que son más o menos las diez películas cuya valoración más en desacuerdo pone a los espectadores:

```
In [95]: rating_std_by_title.sort_values(ascending=False)[:10]
```

```

Out[95]:
title

Dumb & Dumber (1994)           1.321333
Blair Witch Project, The (1999) 1.316368
Natural Born Killers (1994)    1.307198
Tank Girl (1995)                1.277695
Rocky Horror Picture Show, The (1975) 1.260177
Eyes Wide Shut (1999)            1.259624
Evita (1996)                   1.253631
Billy Madison (1995)            1.249970
Fear and Loathing in Las Vegas (1998) 1.246408
Bicentennial Man (1999)          1.245533

Name: rating, dtype: float64

```

Quizá algún lector haya observado que los géneros de las películas se dan como una cadena de texto separada por un carácter |, ya que una película puede pertenecer a varios géneros. Para agrupar los datos de valoraciones por género con mayor facilidad, podemos utilizar el método explode en el dataframe. Veamos cómo funciona esto. Primero, dividimos la cadena de texto genres en una lista de géneros aplicando el método str.split a la serie:

```

In [96]: movies["genres"].head()
Out[96]:
0                               Animation|Children's|Comedy
1                               Adventure|Children's|Fantasy
2                               Comedy|Romance
3                               Comedy|Drama
4                               Comedy

Name: genres, dtype: object

In [97]: movies["genres"].head().str.split("|")
Out[97]:
0                               [Animation, Children's, Comedy]
1                               [Adventure, Children's, Fantasy]
2                               [Comedy, Romance]
3                               [Comedy, Drama]
4                               [Comedy]

Name: genres, dtype: object

```

```

In [98]: movies["genre"] = movies.pop("genres").str.split("|")
In [99]: movies.head()
Out[99]:
movie_id              title \

```

```

0          1                  Toy Story (1995)
1          2                  Jumanji (1995)
2          3      Grumpier Old Men (1995)
3          4      Waiting to Exhale (1995)
4          5 Father of the Bride Part II (1995)

```

genre

```

0          [Animation, Children's, Comedy]
1          [Adventure, Children's, Fantasy]
2          [Comedy, Romance]
3          [Comedy, Drama]
4          [Comedy]

```

Ahora, llamar a `movies.explode("genre")` genera un nuevo dataframe con una fila para cada elemento «interior» de cada lista de géneros de películas. Por ejemplo, si una película está clasificada como comedia y romance, entonces habrá dos filas en el resultado, una con “Comedy” y la otra con “Romance”:

```
In [100]: movies_exploded = movies.explode("genre")
```

```
In [101]: movies_exploded[:10]
Out[101]:
```

	movie_id	title	genre
0	1	Toy Story (1995)	Animation
0	1	Toy Story (1995)	Children's
0	1	Toy Story (1995)	Comedy
1	2	Jumanji (1995)	Adventure
1	2	Jumanji (1995)	Children's
1	2	Jumanji (1995)	Fantasy
2	3	Grumpier Old Men (1995)	Comedy
2	3	Grumpier Old Men (1995)	Romance
3	4	Waiting to Exhale (1995)	Comedy
3	4	Waiting to Exhale (1995)	Drama

Ahora podemos combinar las tres tablas y agrupar por género:

```
In [102]: ratings_with_genre = pd.merge(pd.merge(movies_exploded, ratings), users)
```

```
In [103]: ratings_with_genre.iloc[0]
Out[103]:
```

movie_id	title	genre	user_id
1	Toy Story (1995)	Animation	1

```

rating                                5
timestamp                            978824268
gender                                 F
age                                    1
occupation                           10
zip                                  48067

```

Name: 0, dtype: object

```
In [104]: genre_ratings = (ratings_with_genre.groupby(["genre", "age"]))
```

```
.....:          ["rating"].mean()
.....: .unstack("age"))
```

```
In [105]: genre_ratings[:10]
```

```
Out[105]:
```

	1	18	25	35	45	50
age						\
genre						
Action	3.506385	3.447097	3.453358	3.538107	3.528543	3.611333
Adventure	3.449975	3.408525	3.443163	3.515291	3.528963	3.628163
Animation	3.476113	3.624014	3.701228	3.740545	3.734856	3.780020
Children's	3.241642	3.294257	3.426873	3.518423	3.527593	3.556555
Comedy	3.497491	3.460417	3.490385	3.561984	3.591789	3.646868
Crime	3.710170	3.668054	3.680321	3.733736	3.750661	3.810688
Documentary	3.730769	3.865865	3.946690	3.953747	3.966521	3.908108
Drama	3.794735	3.721930	3.726428	3.782512	3.784356	3.878415
Fantasy	3.317647	3.353778	3.452484	3.482301	3.532468	3.581570
Film-Noir	4.145455	3.997368	4.058725	4.064910	4.105376	4.175401

	56
age	
genre	
Action	3.610709
Adventure	3.649064
Animation	3.756233
Children's	3.621822
Comedy	3.650949
Crime	3.832549
Documentary	3.961538
Drama	3.933465
Fantasy	3.532700
Film-Noir	4.125932

13.3 Nombres de bebés de Estados Unidos entre 1880 y 2010

La Administración del Seguro Social de los Estados Unidos (SSA: *Social Security Administration*) ofrece datos sobre la frecuencia con la que se ponen determinados nombres a

los bebés desde 1880 hasta la actualidad. Hadley Wickham, autor de varios paquetes conocidos de R, emplea este conjunto de datos para ilustrar la manipulación de datos en R.

Para cargar este conjunto hay que limpiar un poco los datos, pero una vez hecho tendremos un dataframe parecido a este:

```
In [4]: names.head(10)
Out[4]:
```

	name	sex	births	year
0	Mary	F	7065	1880
1	Anna	F	2604	1880
2	Emma	F	2003	1880
3	Elizabeth	F	1939	1880
4	Minnie	F	1746	1880
5	Margaret	F	1578	1880
6	Ida	F	1472	1880
7	Alice	F	1414	1880
8	Bertha	F	1320	1880
9	Sarah	F	1288	1880

Hay muchas cosas que podríamos querer hacer con este conjunto de datos:

- Visualizar la proporción de bebés a los que se ha dado un determinado nombre (el nuestro propio u otro) a lo largo del tiempo.
- Determinar el rango relativo de un nombre.
- Determinar los nombres más utilizados cada año o los nombres cuya popularidad ha aumentado o disminuido más.
- Analizar tendencias en los nombres: vocales, consonantes, longitud, diversidad global, cambios en la ortografía, primera y última letra, etc.
- Analizar fuentes externas de tendencias: nombres bíblicos, celebridades, cuestiones demográficas, etc.

Con las herramientas de este libro, muchos de estos tipos de análisis están a nuestro alcance, de modo que iremos viendo algunos de ellos.

En el momento de escribir esto, la SSA ofrece archivos de datos, uno al año, con el número total de nacimientos por cada combinación de sexo y nombre. Se puede descargar el archivo de estos datos sin procesar en esta página web: <http://www.ssa.gov/oact/babynames/limits.html>.

Si esta página ya no existe para cuando este libro esté disponible para su lectura, probablemente se puede buscar por internet. Tras descargar el archivo «National data» denominado `names.zip` y descomprimirlo, tendremos un directorio que contiene una serie de archivos como `yob1880.txt`. Yo utilizo el comando `head` de Unix para consultar las primeras

diez líneas de uno de los archivos (en Windows se puede emplear el comando `more` o abrirlo en un editor de texto):

```
In [106]: !head -n 10 datasets/babynames/yob1880.txt
Mary, F, 7065

Anna, F, 2604
Emma, F, 2003
Elizabeth, F, 1939
Minnie, F, 1746
Margaret, F, 1578
Ida, F, 1472
Alice, F, 1414
Bertha, F, 1320

Sarah, F, 1288
```

Como ya está en formato separado por comas, se puede cargar en un dataframe con `pandas.read_csv`:

```
In [107]: names1880 = pd.read_csv("datasets/babynames/yob1880.txt",
.....           names=["name", "sex", "births"])
```

```
In [108]: names1880
Out[108]:
```

	name	sex	births
0	Mary	F	7065
1	Anna	F	2604
2	Emma	F	2003
3	Elizabeth	F	1939
4	Minnie	F	1746
...
1995	Woodie	M	5
1996	Worthy	M	5
1997	Wright	M	5
1998	York	M	5
1999	Zachariah	M	5

[2000 rows x 3 columns]

Los archivos solo contienen nombres que se han utilizado al menos cinco veces al año, de modo que por simplicidad podemos usar la suma de la columna de nacimientos por sexo como número total de nacimientos de ese año:

```
In [109]: names1880.groupby("sex")["births"].sum()
Out[109]:
```

sex

```
F          90993  
M          110493
```

```
Name: births, dtype: int64
```

Como el conjunto de datos se divide en archivos por año, una de las primeras cosas que hay que hacer es montar todos los datos en un solo dataframe y añadir después un campo year. Para ello se puede usar pandas.concat. Ejecutamos lo siguiente en una celda Jupyter:

```
pieces = []  
for year in range(1880, 2011):  
  
    path = f"datasets/babynames/yob{year}.txt"  
    frame = pd.read_csv(path, names=["name", "sex", "births"])  
    # Añade una columna para el año  
    frame["year"] = year  
    pieces.append(frame)  
  
# Concatena todo en un solo dataframe  
names = pd.concat(pieces, ignore_index=True)
```

Aquí hay un par de cosas que conviene mencionar. Primero, debemos recordar que concat combina los objetos DataFrame por fila de forma predeterminada. Segundo, hay que pasar ignore_index=True porque no nos interesa conservar los números de fila originales devueltos por pandas.read_csv. Por tanto, ahora tenemos un solo dataframe que contiene todos los datos de nombres a lo largo de todos los años:

```
In [111]: names  
Out[111]:
```

		name	sex	births	year
0		Mary	F	7065	1880
1		Anna	F	2604	1880
2		Emma	F	2003	1880
3		Elizabeth	F	1939	1880
4		Minnie	F	1746	1880
...	
1690779		Zymaire	M	5	2010
1690780		Zyonne	M	5	2010
1690781		Zyquarius	M	5	2010
1690782		Zyran	M	5	2010
1690783		Zzyzx	M	5	2010

```
[1690784 rows x 4 columns]
```

Con estos datos ya podemos empezar a agregarlos a nivel de año y sexo utilizando groupby o pivot_table (figura 13.4):

```
In [112]: total_births = names.pivot_table("births", index="year",
.....:             columns="sex", aggfunc=sum)
```

```
In [113]: total_births.tail()
Out[113]:
```

sex	F	M
year		
2006	1896468	2050234
2007	1916888	2069242
2008	1883645	2032310
2009	1827643	1973359
2010	1759010	1898382

```
In [114]: total_births.plot(title="Total births by sex and year")
```

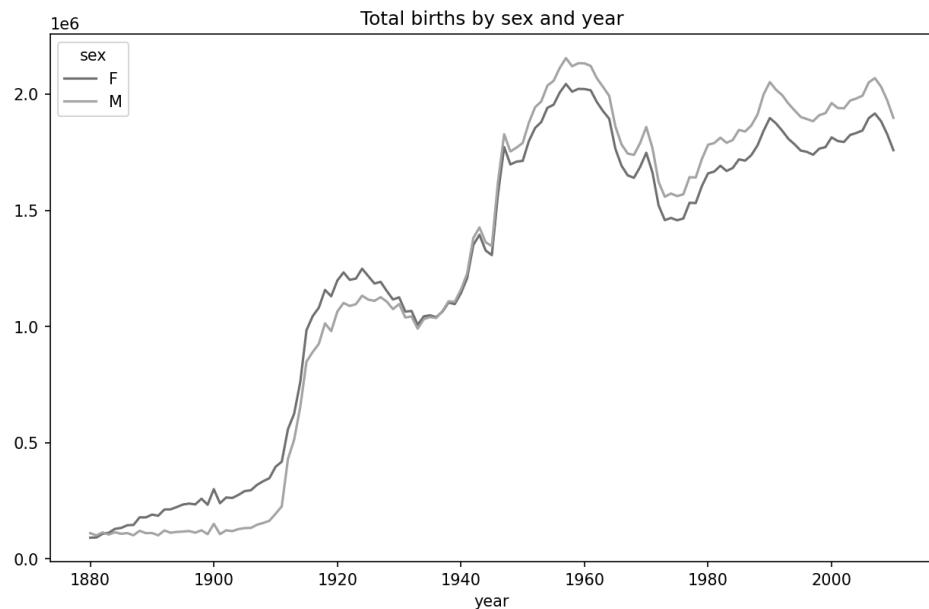


Figura 13.4. Nacimientos totales por sexo y año.

A continuación insertamos una columna prop con la fracción de los bebés a los que se ha dado cada nombre en relación con el número total de nacimientos. Un valor prop de 0.02 indicaría que a 2 de cada 100 bebés se les dio un determinado nombre. Así, agrupamos los datos por año y sexo, y añadimos después la nueva columna a cada grupo:

```
def add_prop(group):
```

```
group[“prop”] = group[“births”] / group[“births”].sum()  
return group
```

```
names = names.groupby([“year”, “sex”]).apply(add_prop)
```

El conjunto de datos resultante tiene ahora las siguientes columnas:

```
In [116]: names  
Out[116]:
```

		name	sex	births	year	prop
0		Mary	F	7065	1880	0.077643
1		Anna	F	2604	1880	0.028618
2		Emma	F	2003	1880	0.022013
3		Elizabeth	F	1939	1880	0.021309
4		Minnie	F	1746	1880	0.019188
...
1690779		Zymaire	M	5	2010	0.000003
1690780		Zyonne	M	5	2010	0.000003
1690781		Zyquarius	M	5	2010	0.000003
1690782		Zyran	M	5	2010	0.000003
1690783		Zzyzx	M	5	2010	0.000003

```
[1690784 rows x 5 columns]
```

Al realizar una operación de grupo como esta, a menudo suele resultar muy valioso hacer una comprobación de salud, algo así como verificar que la columna prop sume 1 dentro de todos los grupos:

```
In [117]: names.groupby([“year”, “sex”])[“prop”].sum()  
Out[117]:  
year sex
```

1880		F	1.0
		M	1.0
1881		F	1.0
		M	1.0
1882		F	1.0
			...
2008		M	1.0
2009		F	1.0
		M	1.0
2010		F	1.0
		M	1.0

```
Name: prop, Length: 262, dtype: float64
```

Ahora que está hecho, voy a extraer un subconjunto de los datos para facilitar la realización de más análisis: los primeros 1000 nombres para cada combinación de sexo y año, operación que de nuevo es de grupo:

```
In [118]: def get_top1000(group):  
    ....:     return group.sort_values("births", ascending=False)[:1000]  
  
In [119]: grouped = names.groupby(["year", "sex"])  
  
In [120]: top1000 = grouped.apply(get_top1000)  
  
In [121]: top1000.head()  
Out[121]:
```

year	sex	name	sex	births	year	prop
1880	F	Mary	F	7065	1880	0.077643
	0	Anna	F	2604	1880	0.028618
	1	Emma	F	2003	1880	0.022013
	2	Elizabeth	F	1939	1880	0.021309
	3	Minnie	F	1746	1880	0.019188
	4					

Podemos quitar el índice de grupo, pues no lo necesitamos para nuestro análisis:

```
In [122]: top1000 = top1000.reset_index(drop=True)
```

El conjunto de datos resultante es ahora bastante más reducido:

```
In [123]: top1000.head()  
Out[123]:
```

	name	sex	births	year	prop
0	Mary	F	7065	1880	0.077643
1	Anna	F	2604	1880	0.028618
2	Emma	F	2003	1880	0.022013
3	Elizabeth	F	1939	1880	0.021309
4	Minnie	F	1746	1880	0.019188

Usaremos este conjunto de datos de los 1000 primeros en las siguientes investigaciones que efectuaremos en los datos.

Análisis de tendencias en los nombres

Teniendo el conjunto de datos completo y el de los 1000 primeros, podemos empezar a analizar varias tendencias en los nombres que pueden resultar de interés. Primero podemos dividir los 1000 primeros nombres en los que son de chico y los que son de chica:

```
In [124]: boys = top1000[top1000["sex"] == "M"]
```

```
In [125]: girls = top1000[top1000["sex"] == "F"]
```

Se pueden crear gráficos de sencillas series temporales, como, por ejemplo, el número de Juanes (*John*) y Marías (*Mary*) por cada año, pero hacen falta ciertas manipulaciones para que resulten más útiles. Formemos una tabla dinámica con el número total de nacimientos por año y nombre:

```
In [126]: total_births = top1000.pivot_table("births", index="year",
.....:                 columns="name",
.....:                 aggfunc=sum)
```

Ahora sí podemos crear un gráfico para unos cuantos nombres con el método `plot` del objeto DataFrame (la figura 13.5 muestra el resultado):

```
In [127]: total_births.info()
<class 'pandas.core.frame.DataFrame'>

Int64Index: 131 entries, 1880 to 2010
Columns: 6868 entries, Aaden to Zuri
dtypes: float64(6868)
memory usage: 6.9 MB

In [128]: subset = total_births[["John", "Harry", "Mary", "Marilyn"]]

In [129]: subset.plot(subplots=True, figsize=(12, 10),
.....:                 title="Number of births per year")
```

Viendo esto podríamos concluir que estos nombres han perdido el atractivo para la población americana. Pero en realidad es mucho más complicado que eso, como veremos en la siguiente sección.

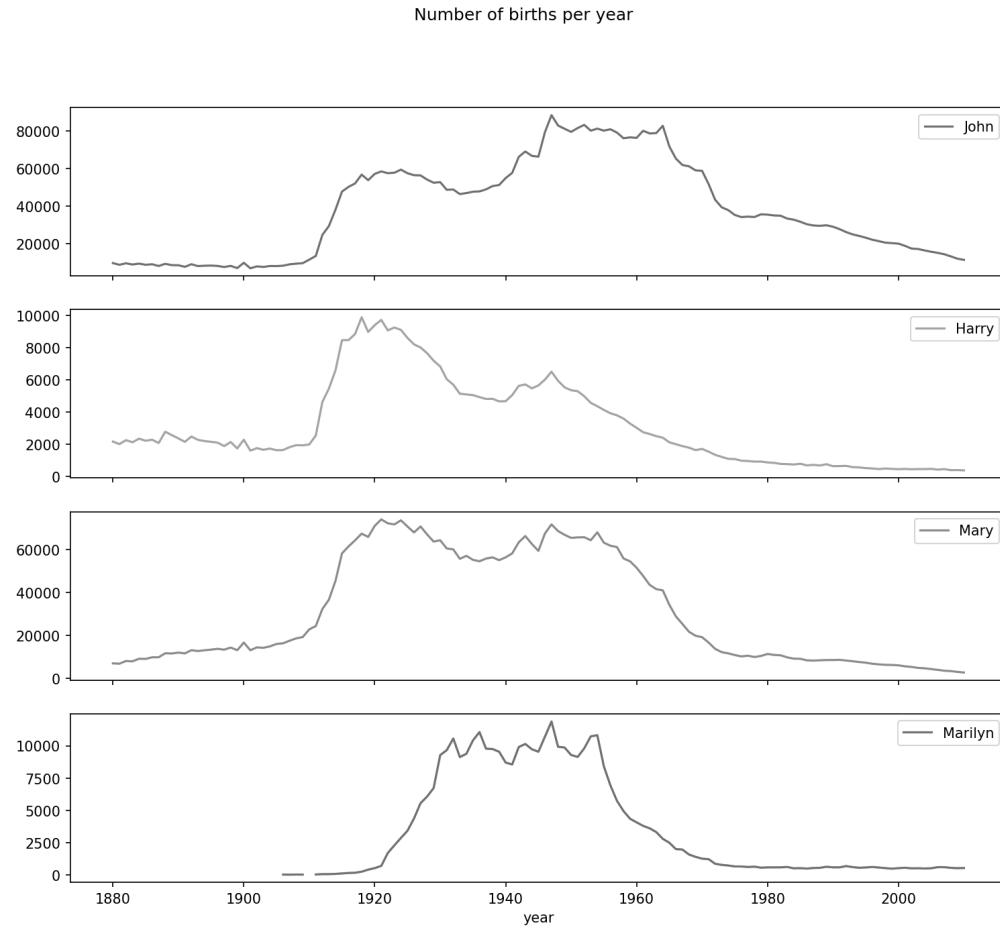


Figura 13.5. Algunos nombres de chico y chica a lo largo del tiempo.

Medición del aumento de la diversidad en los nombres

Una explicación de las disminuciones en los gráficos es que menos padres están eligiendo nombres comunes para sus hijos. Esta hipótesis puede explorarse y confirmarse en los datos. Una medida es la proporción de nacimientos representada por los 1000 primeros nombres más utilizados, que agrego y visualizo por año y sexo (en el gráfico de la figura 13.6):

```
In [131]: table = top1000.pivot_table("prop", index="year",
.....:             columns="sex", aggfunc=sum)

In [132]: table.plot(title="Sum of table1000.prop by year and sex",
.....:             yticks=np.linspace(0, 1.2, 13))
```



Figura 13.6. Proporción de nacimientos representada en los primeros 1000 nombres por sexo.

Se puede observar que, de hecho, parece haber una mayor diversidad en los nombres (disminuyendo al mismo tiempo la proporción total en los primeros 1000). Otra métrica interesante es el número de nombres distintos, tomados en orden de popularidad de mayor a menor, en el 50 % de los nacimientos. Este número es más difícil de calcular. Veamos solamente los nombres de chico de 2010:

```
In [133]: df = boys[boys["year"] == 2010]
```

```
In [134]: df
```

```
Out[134]:
```

	name	sex	births	year	prop
260877	Jacob	M	21875	2010	0.011523
260878	Ethan	M	17866	2010	0.009411
260879	Michael	M	17133	2010	0.009025
260880	Jayden	M	17030	2010	0.008971
260881	William	M	16870	2010	0.008887
...
261872	Camilo	M	194	2010	0.000102
261873	Destin	M	194	2010	0.000102
261874	Jaquan	M	194	2010	0.000102
261875	Jaydan	M	194	2010	0.000102
261876	Maxton	M	193	2010	0.000102

```
[1000 rows x 5 columns]
```

Tras ordenar `prop` en orden descendente, queremos saber cuántos de los nombres más utilizados hacen falta para llegar al 50 %. Podríamos escribir un bucle `for` para esto, pero utilizar un array NumPy vectorizado es mucho más eficaz computacionalmente hablando. Tomando la suma acumulativa `cumsum` de `prop` y llamando después al método `searchsorted`, se obtiene la posición de la suma acumulativa en la que se necesitaría insertar 0.5 para mantenerlo en orden:

```
In [135]: prop_cumsum = df[“prop”].sort_values(ascending=False).cumsum()
```

```
In [136]: prop_cumsum[:10]
```

```
Out[136]:
```

260877	0.011523
260878	0.020934
260879	0.029959
260880	0.038930
260881	0.047817
260882	0.056579
260883	0.065155
260884	0.073414
260885	0.081528
260886	0.089621

```
Name: prop, dtype: float64
```

```
In [137]: prop_cumsum.searchsorted(0.5)
```

```
Out[137]: 116
```

Como los arrays están indexados al cero, sumar 1 a este resultado nos da 117. Sin embargo, en 1900 este número era mucho menor:

```
In [138]: df = boys[boys.year == 1900]
```

```
In [139]: in1900 = df.sort_values(“prop”, ascending=False).prop.cumsum()
```

```
In [140]: in1900.searchsorted(0.5) + 1
```

```
Out[140]: 25
```

Ahora podemos aplicar esta operación a cada combinación de año y sexo, aplicar `groupby` a estos campos y después una función que devuelve el recuento para cada grupo:

```
def get_quantile_count(group, q=0.5):
    group = group.sort_values(“prop”, ascending=False)

    return group.prop.cumsum().searchsorted(q) + 1
diversity = top1000.groupby([“year”, “sex”]).apply(get_quantile_count)

diversity = diversity.unstack()
```

Este dataframe resultante `diversity` tiene ahora dos series temporales, una para cada sexo, indexadas por año, que también se pueden inspeccionar y visualizar como antes (véase la

figura 13.7):

```
In [143]: diversity.head()  
Out[143]:
```

sex	F	M
year		
1880	38	14
1881	38	14
1882	38	15
1883	39	15
1884	39	16

```
In [144]: diversity.plot(title="Number of popular names in top 50%")
```

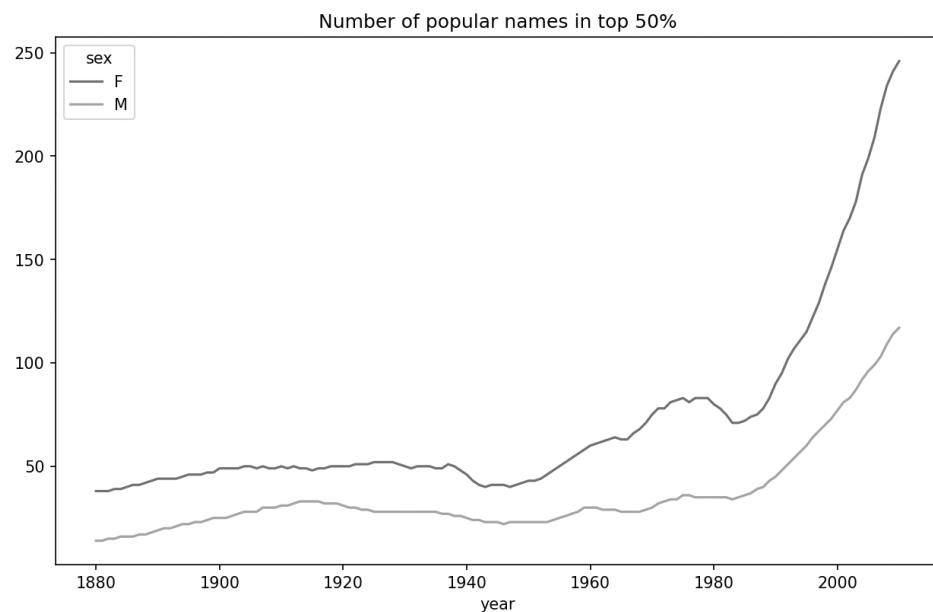


Figura 13.7. Gráfico de métrica de diversidad por año.

Como se puede observar, los nombres de chica siempre han sido más diversos que los de chico, simplemente ha ocurrido que con el tiempo lo han sido más. Queda al albedrío del lector realizar más análisis para averiguar lo que está propiciando exactamente la diversidad, como, por ejemplo, el aumento de ortografías alternativas.

La revolución de la «última letra»

En 2007, la investigadora de nombres de bebés Laura Wattenberg averiguó que la distribución de nombres de chico por la letra final ha cambiado de manera significativa en los últimos 100 años. Para comprobarlo, primero agregamos todos los nacimientos del conjunto de datos completo por año, sexo y letra final:

```

def get_last_letter(x):

    return x[-1]

last_letters = names[“name”].map(get_last_letter)
last_letters.name = “last_letter”
table = names.pivot_table(“births”, index=last_letters,
                           columns=[“sex”, “year”], aggfunc=sum)

```

Después seleccionamos tres años representativos a lo largo de este período y visualizamos las primeras filas:

```
In [146]: subtable = table.reindex(columns=[1910, 1960, 2010],
                                   level=“year”)
```

```
In [147]: subtable.head()
Out[147]:
```

sex	F			M		
year	1910	1960	2010	1910	1960	2010
last_letter						
a	108376.0	691247.0	670605.0	977.0	5204.0	28438.0
b	NaN	694.0	450.0	411.0	3912.0	38859.0
c	5.0	49.0	946.0	482.0	15476.0	23125.0
d	6750.0	3729.0	2607.0	22111.0	262112.0	44398.0
e	133569.0	435013.0	313833.0	28655.0	178823.0	129012.0

Ahora normalizamos la tabla por nacimientos totales para obtener una nueva tabla que contenga la proporción de nacimientos totales por cada sexo que termina en cada letra:

```
In [148]: subtable.sum()
Out[148]:
```

sex	year	
F	1910	396416.0
	1960	2022062.0
	2010	1759010.0
M	1910	194198.0
	1960	2132588.0
	2010	1898382.0

```
dtype: float64
```

```
In [149]: letter_prop = subtable / subtable.sum()
```

```
In [150]: letter_prop
Out[150]:
```

sex	F			M		
year	1910	1960	2010	1910	1960	2010
last_letter						
a	0.273390	0.341853	0.381240	0.005031	0.002440	0.014980
b		NaN	0.000343	0.000256	0.002116	0.001834
c	0.000013	0.000024	0.000538	0.002482	0.007257	0.012181
d	0.017028	0.001844	0.001482	0.113858	0.122908	0.023387
e	0.336941	0.215133	0.178415	0.147556	0.083853	0.067959
...
v		NaN	0.000060	0.000117	0.000113	0.000037
w	0.000020	0.000031	0.001182	0.006329	0.007711	0.016148
x	0.000015	0.000037	0.000727	0.003965	0.001851	0.008614
y	0.110972	0.152569	0.116828	0.077349	0.160987	0.058168
z	0.002439	0.000659	0.000704	0.000170	0.000184	0.001831

[26 rows x 6 columns]

Teniendo las proporciones de letra, podemos crear gráficos de barras para cada sexo, divididos por año (figura 13.8):

```
import matplotlib.pyplot as plt
fig, axes = plt.subplots(2, 1, figsize=(10, 8))
letter_prop["M"].plot(kind="bar", rot=0, ax=axes[0], title="Male")
letter_prop["F"].plot(kind="bar", rot=0, ax=axes[1], title="Female",
                      legend=False)
```

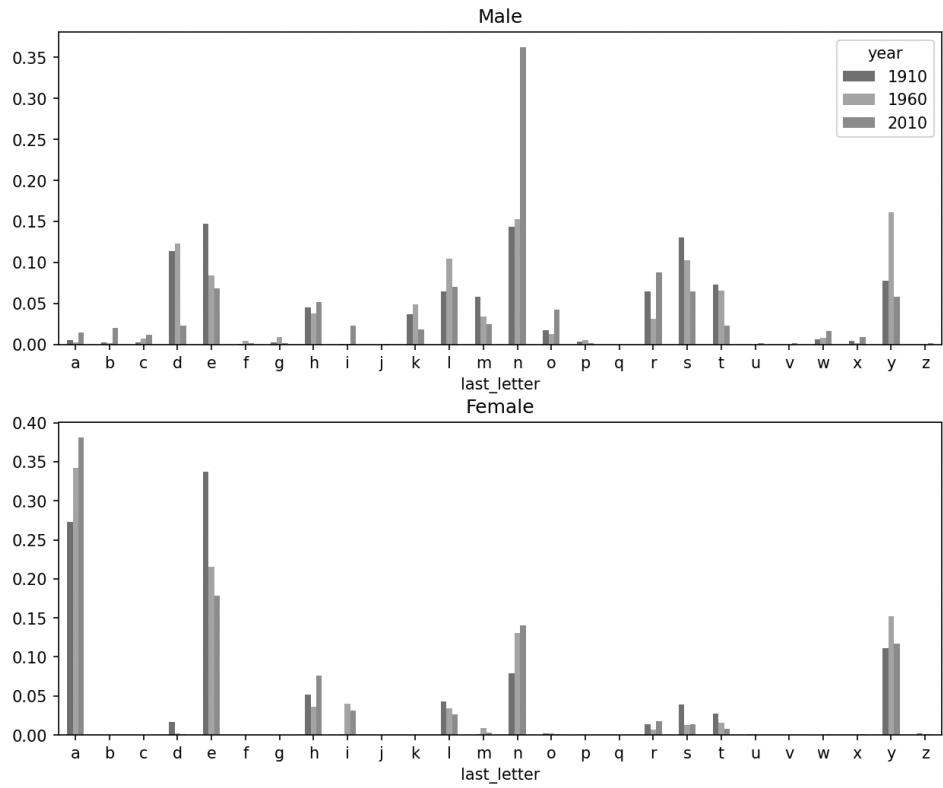


Figura 13.8. Proporción de nombres de chico y chica que terminan por cada letra.

Como se puede comprobar, la utilización de nombres de chico que terminan en n ha experimentado un importante crecimiento desde los años 60. Volviendo a la tabla completa creada antes, vuelvo a normalizar por año y sexo y selecciono un subconjunto de letras para los nombres de chico, transponiendo finalmente para convertir cada columna en una serie temporal:

```
In [153]: letter_prop = table / table.sum()

In [154]: dny_ts = letter_prop.loc[["d", "n", "y"], "M"].T
```

```
In [155]: dny_ts.head()
Out[155]:
```

last_letter	d	n	y
year			
1880	0.083055	0.153213	0.075760
1881	0.083247	0.153214	0.077451
1882	0.085340	0.149560	0.077537
1883	0.084066	0.151646	0.079144
1884	0.086120	0.149915	0.080405

Con este objeto DataFrame de series temporales, podemos crear de nuevo un gráfico de las tendencias a lo largo del tiempo con su método `plot` (véase la figura 13.9):

```
In [158]: dny_ts.plot()
```

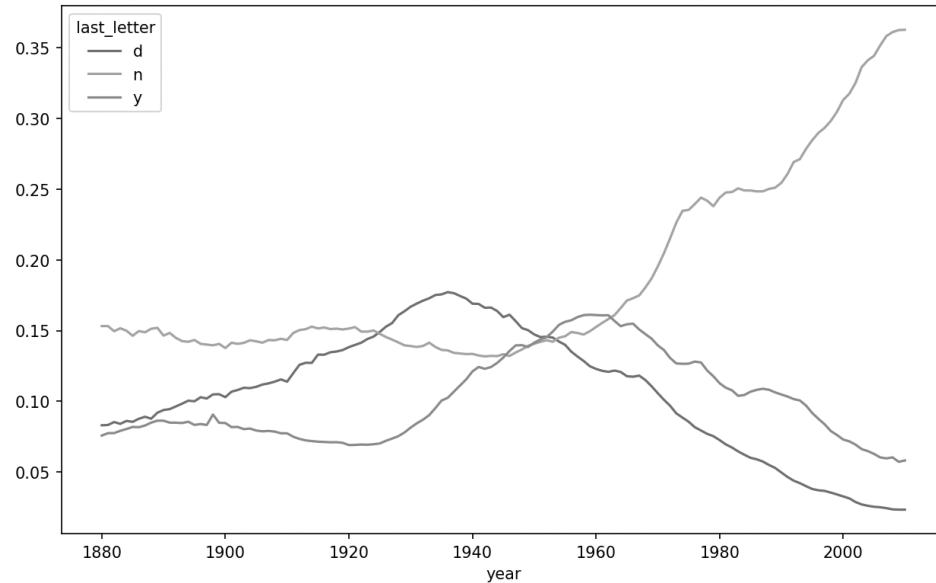


Figura 13.9. Proporción de chicos nacidos con nombres que terminan en d, n e y a lo largo del tiempo.

Nombres de chico que se convirtieron en nombres de chica (y viceversa)

Otra tendencia divertida es averiguar los nombres que se utilizaban antes más para un sexo, pero que con el tiempo se han convertido en preferidos para el otro. Un ejemplo es el nombre Lesley o Leslie. Volviendo al objeto DataFrame `top1000`, calculo una lista de nombres que aparecen en el conjunto de datos y que empiezan por «Lesl»:

```
In [159]: all_names = pd.Series(top1000["name"].unique())
In [160]: lesley_like = all_names[all_names.str.contains("Lesl")]
In [161]: lesley_like
Out[161]:
```

632	Leslie
2294	Lesley
4262	Leslee
4728	Lesli
6103	Lesly

```
dtype: object
```

A partir de aquí podemos filtrar para quedarnos solo con esos nombres y sumar los nacimientos agrupados por nombre para ver las frecuencias relativas:

```
In [162]: filtered = top1000[top1000["name"].isin(lesley_like)]
```

```
In [163]: filtered.groupby("name")["births"].sum()
```

```
Out[163]:
```

```
name
```

Leslee	1082
Lesley	35022
Lesli	929
Leslie	370429
Lesly	10067

```
Name: births, dtype: int64
```

A continuación, agreguemos por sexo y año, y normalicemos dentro del año:

```
In [164]: table = filtered.pivot_table("births", index="year",
```

```
.....:           columns="sex", aggfunc="sum")
```

```
In [165]: table = table.div(table.sum(axis="columns"), axis="index")
```

```
In [166]: table.tail()
```

```
Out[166]:
```

	F	M
sex		
year		
2006	1.0	NaN
2007	1.0	NaN
2008	1.0	NaN
2009	1.0	NaN
2010	1.0	NaN

Finalmente, ahora es posible crear un gráfico del desglose por sexo a lo largo del tiempo (figura 13.10):

```
In [168]: table.plot(style={"M": "k-", "F": "k-"}))
```



Figura 13.10. Proporción de nombres parecidos a Lesley de chico y chica a lo largo del tiempo.

13.4 Base de datos de alimentos del USDA

El Departamento de Agricultura de los Estados Unidos (*United States Department of Agriculture*, USDA) ofrece una base de datos de información sobre los nutrientes de los alimentos. La programadora Ashley Williams creó una versión de esta base de datos en formato JSON. Los registros tienen este aspecto:

```
{
  "id": 21441,
  "description": "KENTUCKY FRIED CHICKEN, Fried Chicken, EXTRA CRISPY, Wing, meat and skin with breading",
  "tags": ["KFC"],
  "manufacturer": "Kentucky Fried Chicken",
  "group": "Fast Foods",
  "portions": [
    {
      "amount": 1,
      "unit": "wing, with skin",
      "grams": 68.0
    },
    ...
  ],
  "nutrients": [
    {
      "value": 20.8,
      "units": "g",
    }
  ]
}
```

```

    "description": "Protein",
    "group": "Composition"
},
...
]
}

```

Cada alimento tiene distintos atributos que lo identifican, además de dos listas de nutrientes y tamaños de porción. Los datos organizados de este modo no son especialmente susceptibles de análisis, por lo cual debemos trabajar un poco para darles un mejor formato.

Se puede cargar este archivo en Python con cualquier librería JSON. Yo voy a utilizar el módulo `json` interno de Python:

```

In [169]: import json
In [170]: db = json.load(open("datasets/usda_food/database.json"))
In [171]: len(db)
Out[171]: 6636

```

Cada entrada de `db` es un diccionario que contiene todos los datos para cada alimento. El campo “`nutrients`” es una lista de diccionarios, uno por cada nutriente:

```

In [172]: db[0].keys()
Out[172]: dict_keys(['id', 'description', 'tags', 'manufacturer', 'group',
'portions', 'nutrients'])
In [173]: db[0]["nutrients"][0]
Out[173]:
{'value': 25.18,
'units': 'g',
'description': 'Protein',
'group': 'Composition'}
In [174]: nutrients = pd.DataFrame(db[0]["nutrients"])
In [175]: nutrients.head(7)
Out[175]:

```

	value	units	description	group
0	25.18	g	Protein	Composition
1	29.20	g	Total lipid (fat)	Composition
2	3.06	g	Carbohydrate, by difference	Composition
3	3.28	g	Ash	Other
4	376.00	kcal	Energy	Energy
5	39.28	g	Water	Composition
6	1573.00	kJ	Energy	Energy

Al convertir una lista de diccionarios en un dataframe, podemos especificar la lista de campos que queremos extraer. Tomaremos los nombres de los alimentos, el grupo, el identificador y el fabricante:

```
In [176]: info_keys = ["description", "group", "id", "manufacturer"]
```

```
In [177]: info = pd.DataFrame(db, columns=info_keys)
```

```
In [178]: info.head()
```

```
Out[178]:
```

```
          description      group   id \
0        Cheese, caraway Dairy and Egg 1008
1        Cheese, cheddar  Dairy and Egg 1009
2        Cheese, edam    Dairy and Egg 1018
3        Cheese, feta    Dairy and Egg 1019
4  Cheese, mozzarella,  Dairy and Egg 1028
   part skim milk      Products          
```

manufacturer

```
0
1
2
3
4
```

```
In [179]: info.info()
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 6636 entries, 0 to 6635
Data columns (total 4 columns):
```

#	Column	Non-Null	Count	Dtype
0	description	6636	non-null	object
1	group	6636	non-null	object
2	id	6636	non-null	int64
3	manufacturer	5195	non-null	object

```
dtypes: int64(1), object(3)
memory usage: 207.5+ KB
```

Por el resultado de `info.info()`, podemos ver que hay datos ausentes en la columna `manufacturer`.

Se puede observar la distribución de grupos de alimentos con `value_counts`:

```
In [180]: pd.value_counts(info["group"])[10]
```

```
Out[180]:
```

Vegetables and Vegetable Products	812
Beef Products	618
Baked Products	496
Breakfast Cereals	403
Legumes and Legume Products	365
Fast Foods	365
Lamb, Veal, and Game Products	345
Sweets	341
Fruits and Fruit Juices	328
Pork Products	328

```
Name: group, dtype: int64
```

Ahora, para analizar todos los datos de nutrientes, es más fácil organizar los nutrientes para cada alimento en una sola tabla grande. Pero para ello necesitamos dar varios pasos previos. Primero, hay que convertir cada lista de nutrientes de alimentos en un dataframe, añadir una columna para el `id` del alimento y añadir el dataframe a una lista. Después, se pueden concatenar con `concat`. Ejecutamos el siguiente código en una celda Jupyter:

```
nutrients = []
for rec in db:
    fnuts = pd.DataFrame(rec["nutrients"])
    fnuts["id"] = rec["id"]
    nutrients.append(fnuts)

nutrients = pd.concat(nutrients, ignore_index=True)
```

Si todo ha ido bien, `nutrients` debe verse de este modo:

```
In [182]: nutrients
Out[182]:
```

```
      value units          description      group   id
0     25.180   g            Protein Composition  1008
1     29.200   g        Total lipid (fat) Composition  1008
2     3.060   g  Carbohydrate, by difference Composition  1008
3     3.280   g             Ash           Other  1008
4    376.000  kcal            Energy           Energy  1008
...       ...   ...
389350  0.000  mcg  Vitamin B-12, added  Vitamins 43546
389351  0.000   mg        Cholesterol      Other 43546
389352  0.072   g  Fatty acids, total saturated      Other 43546
389353  0.028   g  Fatty acids, total monounsaturated      Other 43546
389354  0.041   g  Fatty acids, total polyunsaturated      Other   546
```

```
[389355 rows x 5 columns]
```

Me di cuenta de que hay duplicados en este dataframe, de modo que quitarlos facilita las cosas:

```
In [183]: nutrients.duplicated().sum() # número de duplicados  
Out[183]: 14179
```

```
In [184]: nutrients = nutrients.drop_duplicates()
```

Como “group” y “description” están en ambos objetos DataFrame, podemos renombrarlos para mayor claridad:

```
In [185]: col_mapping = {"description" : "food",  
.....: "group" : "fgroup"}
```

```
In [186]: info = info.rename(columns=col_mapping, copy=False)
```

```
In [187]: info.info()  
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 6636 entries, 0 to 6635  
Data columns (total 4 columns):
```

#	Column	Non-Null	Count	Dtype
0	food	6636	non-null	object
1	fgroup	6636	non-null	object
2	id	6636	non-null	int64
3	manufacturer	5195	non-null	object

```
dtypes: int64(1), object(3)
```

```
memory usage: 207.5+ KB
```

```
In [188]: col_mapping = {"description" : "nutrient",  
.....: "group" : "nutgroup"}
```

```
In [189]: nutrients = nutrients.rename(columns=col_mapping, copy=False)
```

```
In [190]: nutrients  
Out[190]
```

	value	units		nutrient	nutgroup	id
0	25.180	g		Protein	Composition	1008
1	29.200	g	Total lipid (fat)	Composition	1008	
2	3.060	g	Carbohydrate, by difference	Composition	1008	
3	3.280	g		Ash	Other	1008
4	376.000	kcal		Energy	Energy	1008
...
389350	0.000	mcg	Vitamin B-12, added	Vitamins		43546

```

389351    0.000    mg                               Cholesterol      Other 43546
389352    0.072    g       Fatty acids, total saturated  Other 43546
389353    0.028    g   Fatty acids, total monounsaturated  Other 43546
389354    0.041    g   Fatty acids, total polyunsaturated  Other 43546

```

[375176 rows x 5 columns]

Con todo esto hecho, estamos listos para combinar info con nutrients:

```
In [191]: ndata = pd.merge(nutrients, info, on="id")
```

```
In [192]: ndata.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 375176 entries, 0 to 375175
Data columns (total 8 columns):
```

#	Column	Non-Null	Count	Dtype
0	value	375176	non-null	float64
1	units	375176	non-null	object
2	nutrient	375176	non-null	object
3	nutgroup	375176	non-null	object
4	id	375176	non-null	int64
5	food	375176	non-null	object
6	fgroup	375176	non-null	object
7	manufacturer	293054	non-null	object

```
dtypes: float64(1), int64(1), object(6)
memory usage: 25.8+ MB
```

```
In [193]: ndata.iloc[30000]
Out[193]:
```

```

value          0.04
units           g
nutrient        Glycine
nutgroup        Amino Acids
id              6158
food            Soup, tomato bisque, canned, condensed
fgroup          Soups, Sauces, and Gravies

manufacturer
Name: 30000, dtype: object

```

Ahora podríamos crear un gráfico de valores promedio por grupo de alimento y tipo de nutriente (véase la figura 13.11):

```
In [195]: result      =      ndata.groupby(["nutrient",      "fgroup"])
["value"].quantile(0.5)
```

```
In [196]: result["Zinc, Zn"].sort_values().plot(kind="barh")
```

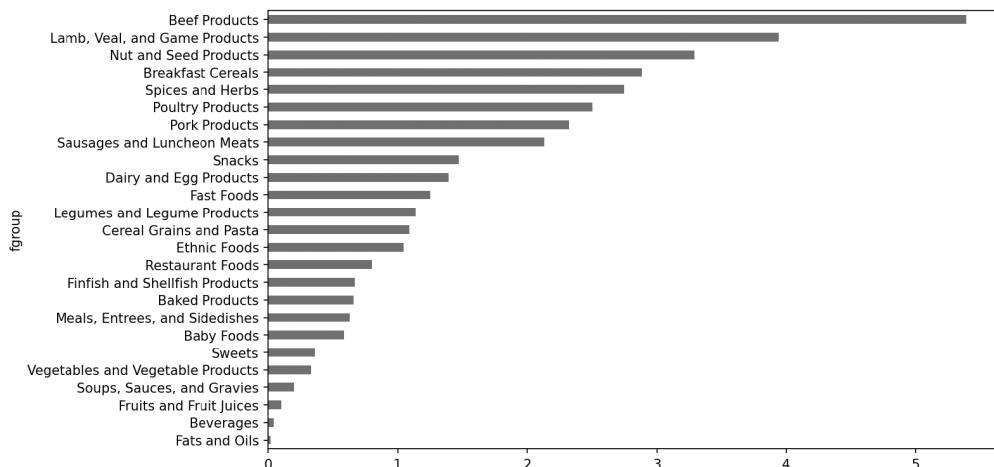


Figura 13.11. Valores medios de zinc por grupo de alimentos.

Utilizando los métodos `idxmax` o `argmax` del objeto Series, podemos averiguar qué alimento es más denso en cada nutriente. Ejecutamos lo siguiente en una celda Jupyter:

```
by_nutrient = ndata.groupby(["nutgroup", "nutrient"])
def get_maximum(x):
    return x.loc[x.value.idxmax()]
max_foods = by_nutrient.apply(get_maximum)[["value", "food"]]
# hace food un poco más pequeño

max_foods["food"] = max_foods["food"].str[:50]
```

El objeto DataFrame resultante es demasiado grande para mostrarlo en el libro; este es tan solo el grupo de nutrientes “Amino Acids”:

```
In [198]: max_foods.loc["Amino Acids"]["food"]
Out[198]:
```

nutrient	Food
Alanine	Gelatins, dry powder, unsweetened
Arginine	Seeds, sesame flour, low-fat
Aspartic acid	Soy protein isolate
Cystine	Seeds, cottonseed flour, low fat (glandless)
Glutamic acid	Soy protein isolate
Glycine	Gelatins, dry powder, unsweetened
Histidine	Whale, beluga, meat, dried (Alaska Native)
Hydroxyproline	KENTUCKY FRIED CHICKEN, Fried Chicken, ORIGINAL RE
Isoleucine	Soy protein isolate, PROTEIN TECHNOLOGIES INTERNAT
Leucine	Soy protein isolate, PROTEIN TECHNOLOGIES INTERNAT
Lysine	Seal, bearded (Oogruck), meat, dried (Alaska Native)
Methionine	Fish, cod, Atlantic, dried and salted
Phenylalanine	Soy protein isolate, PROTEIN TECHNOLOGIES INTERNAT

Proline	Gelatins, dry powder, unsweetened
Serine	Soy protein isolate, PROTEIN TECHNOLOGIES INTERNAT
Threonine	Soy protein isolate, PROTEIN TECHNOLOGIES INTERNAT
Tryptophan	Sea lion, Steller, meat with fat (Alaska Native)
Tyrosine	Soy protein isolate, PROTEIN TECHNOLOGIES INTERNAT
Valine	Soy protein isolate, PROTEIN TECHNOLOGIES INTERNAT

Name: food, dtype: object

13.5 Base de datos de la Comisión de Elecciones Federales de 2012

La Comisión de Elecciones Federales de Estados Unidos (*Federal Election Commission*, FEC) publica datos sobre contribuciones a las campañas políticas. Entre ellos se incluyen los nombres de los contribuyentes, su ocupación y empleador, su dirección y la cantidad contribuida. Los datos de las contribuciones en las elecciones presidenciales de 2012 estaban contenidos en un solo archivo CSV de 150 Mb denominado P00000001-ALL.csv (más información en el repositorio de datos del libro), que puede cargarse con pandas.read_csv:

```
In [199]: fec = pd.read_csv("datasets/fec/P00000001-ALL.csv",
low_memory=False)
```

```
In [200]: fec.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1001731 entries, 0 to 1001730
Data columns (total 16 columns):
```

#	Column	Non-Null	Count	Dtype
--	--	--	--	
0	cmte_id	1001731	non-null	object
1	cand_id	1001731	non-null	object
2	cand_nm	1001731	non-null	object
3	contbr_nm	1001731	non-null	object
4	contbr_city	1001712	non-null	object
5	contbr_st	1001727	non-null	object
6	contbr_zip	1001620	non-null	object
7	contbr_employer	988002	non-null	object
8	contbr_occupation	993301	non-null	object
9	contb_receipt_amt	1001731	non-null	float64
10	contb_receipt_dt	1001731	non-null	object
11	receipt_desc	14166	non-null	object
12	memo_cd	92482	non-null	object
13	memo_text	97770	non-null	object
14	form_tp	1001731	non-null	object
15	file_num	1001731	non-null	int64

```
dtypes: float64(1), int64(1), object(14)
memory usage: 122.3+ MB
```



Varias personas me pidieron que actualizara el conjunto de datos de las elecciones de 2012 a las de 2016 o 2020. Lamentablemente, los conjuntos de datos más recientes proporcionados por la FEC son cada vez más grandes y complejos, por eso decidí que trabajar con ellos aquí nos distraería de las técnicas de análisis que es mi intención ilustrar.

Un registro de ejemplo del objeto DataFrame es algo así:

```
In [201]: fec.iloc[123456]  
Out[201]:
```

```
cmte_id           C00431445  
cand_id          P80003338  
cand_nm          Obama, Barack  
contbr_nm         ELLMAN, IRA  
contbr_city       TEMPE  
contbr_st         AZ  
contbr_zip        852816719  
contrbr_employer ARIZONA STATE UNIVERSITY  
contrbr_occupation PROFESSOR  
contb_receipt_amt 50.0  
contb_receipt_dt  01-DEC-11  
receipt_desc      NaN  
memo_cd           NaN  
memo_text         NaN  
form_tp           SA17A  
file_num          772372  
  
Name: 123456, dtype: object
```

Podemos pensar en varias formas de empezar a segmentar estos datos para extraer estadísticas informativas sobre los donantes y los patrones en las contribuciones a las campañas. Voy a mostrar distintos análisis que aplican las técnicas aprendidas en este libro.

Podemos ver que en los datos no hay afiliaciones a partidos políticos, de modo que sería útil añadirlos. Es posible obtener una lista de todos los candidatos únicos empleando `unique`:

```
In [202]: unique_cands = fec["cand_nm"].unique()  
  
In [203]: unique_cands  
Out[203]:  
array(['Bachmann, Michelle', 'Romney, Mitt', 'Obama, Barack',  
       'Roemer, Charles E. \'Buddy\' III', 'Pawlenty, Timothy',  
       'Johnson, Gary Earl', 'Paul, Ron', 'Santorum, Rick',  
       'Cain, Herman', 'Gingrich, Newt', 'McCotter, Thaddeus G',  
       'Huntsman, Jon', 'Perry, Rick'], dtype=object)  
  
In [204]: unique_cands[2]  
Out[204]: 'Obama, Barack'
```

Una forma de indicar afiliación a un partido es mediante un diccionario³:

```
parties = {"Bachmann, Michelle": "Republican",
           "Cain, Herman": "Republican",
           "Gingrich, Newt": "Republican",
           "Huntsman, Jon": "Republican",
           "Johnson, Gary Earl": "Republican",
           "McCotter, Thaddeus G": "Republican",
           "Obama, Barack": "Democrat",
           "Paul, Ron": "Republican",
           "Pawlenty, Timothy": "Republican",
           "Perry, Rick": "Republican",
           "Roemer, Charles E. 'Buddy' III": "Republican",
           "Romney, Mitt": "Republican",
           "Santorum, Rick": "Republican"}
```

Ahora, utilizando este mapeado y el método `map` de los objetos Series, podemos calcular un array de partidos políticos a partir de los nombres de los candidatos:

```
In [206]: fec["cand_nm"][123456:123461]
Out[206]:
```

123456	Obama, Barack
123457	Obama, Barack
123458	Obama, Barack
123459	Obama, Barack
123460	Obama, Barack

```
Name: cand_nm, dtype: object
```

```
In [207]: fec["cand_nm"][123456:123461].map(parties)
Out[207]:
```

123456	Democrat
123457	Democrat
123458	Democrat
123459	Democrat
123460	Democrat

```
Name: cand_nm, dtype: object
```

```
# Lo añade como columna
```

```
In [208]: fec["party"] = fec["cand_nm"].map(parties)
```

```
In [209]: fec["party"].value_counts()
Out[209]:
```

```
Democrat          593746
Republican        407985
```

```
Name: party, dtype: int64
```

Ahora veamos un par de puntos preparatorios. Primero, estos datos incluyen contribuciones y reembolsos (donaciones negativas):

```
In [210]: (fec["contb_receipt_amt"] > 0).value_counts()
Out[210]:
```

True	991475
False	10256

```
Name: contb_receipt_amt, dtype: int64
```

Para simplificar el análisis, restringiré el conjunto de datos a las contribuciones positivas:

```
In [211]: fec = fec[fec["contb_receipt_amt"] > 0]
```

Como Barack Obama y Mitt Romney fueron los dos candidatos principales, también prepararé un subconjunto que incluya solamente las contribuciones a sus campañas:

```
In [212]: fec_mrbo = fec[fec["cand_nm"].isin(["Obama, Barack", "Romney, Mitt"])]
```

Estadísticas de donación por ocupación y empleador

Las donaciones por ocupación son otra estadística frecuentemente estudiada. Por ejemplo, los abogados tienden a donar más dinero a los demócratas, mientras que los ejecutivos de empresas suelen donar más a los republicanos. No es que lo diga yo; los datos hablan por sí solos. En primer lugar, se puede calcular el número total de donaciones por ocupación con `value_counts`:

```
In [213]: fec["contbr_occupation"].value_counts()[:10]
Out[213]:
```

RETIRED	233990
INFORMATION REQUESTED	35107
ATTORNEY	34286
HOMEMAKER	29931
PHYSICIAN	23432
INFORMATION REQUESTED PER BEST EFFORTS	21138
ENGINEER	14334
TEACHER	13990
CONSULTANT	13273
PROFESSOR	12555

```
Name: contbr_occupation, dtype: int64
```

Mirando las ocupaciones se puede notar que muchas se refieren al mismo tipo de trabajo básico, o bien que hay distintas variantes de lo mismo. El siguiente fragmento de código ilustra una técnica para limpiar algunos de ellos mapeando de una ocupación a otra; aquí aplicamos el «truco» de utilizar dict.get para permitir que «cuelen» las ocupaciones sin mapeado:

```
occ_mapping = {  
  
    "INFORMATION REQUESTED PER BEST EFFORTS" : "NOT PROVIDED",  
    "INFORMATION REQUESTED" : "NOT PROVIDED",  
    "INFORMATION REQUESTED (BEST EFFORTS)" : "NOT PROVIDED",  
    "C.E.O." : "CEO"  
  
}  
  
def get_occ(x):  
  
    # Si no hay mapeado, devuelve x  
    return occ_mapping.get(x, x)  
  
fec["contbr_occupation"] = fec["contbr_occupation"].map(get_occ)
```

También hago lo mismo para los empleadores:

```
emp_mapping = {  
  
    "INFORMATION REQUESTED PER BEST EFFORTS" : "NOT PROVIDED",  
    "INFORMATION REQUESTED" : "NOT PROVIDED",  
    "SELF" : "SELF-EMPLOYED",  
    "SELF EMPLOYED" : "SELF-EMPLOYED",  
  
}  
  
def get_emp(x):  
    # Si no se ofrece mapeado, devuelve x  
  
    return emp_mapping.get(x, x)  
  
fec["contbr_employer"] = fec["contbr_employer"].map(f)
```

Ahora podemos usar pivot_table para agregar los datos por partido y ocupación, y después filtrar al subconjunto que donó al menos dos millones de dólares en total:

```
In [216]: by_occupation = fec.pivot_table("contb_receipt_amt",  
.....:           index="contbr_occupation",
```

```

.....:           columns="party", aggfunc="sum")

In [217]: over_2mm = by_occupation[by_occupation.sum(axis="columns") >
2000000]

In [218]: over_2mm
Out[218]:

```

party	Democrat	Republican
contbr_occupation		
ATTORNEY	11141982.97	7477194.43
CEO	2074974.79	4211040.52
CONSULTANT	2459912.71	2544725.45
ENGINEER	951525.55	1818373.70
EXECUTIVE	1355161.05	4138850.09
HOMEMAKER	4248875.80	13634275.78
INVESTOR	884133.00	2431768.92
LAWYER	3160478.87	391224.32
MANAGER	762883.22	1444532.37
NOT PROVIDED	4866973.96	20565473.01
OWNER	1001567.36	2408286.92
PHYSICIAN	3735124.94	3594320.24
PRESIDENT	1878509.95	4720923.76
PROFESSOR	2165071.08	296702.73
REAL ESTATE	528902.09	1625902.25
RETIRED	25305116.38	23561244.49
SELF-EMPLOYED	672393.40	1640252.54

Quizá sea más fácil visualizar estos datos como un gráfico de barras (“barh” significa gráfico de barras horizontales; véase la figura 13.12):

```
In [220]: over_2mm.plot(kind="barh")
```

Quizá nos interesen las ocupaciones principales de los donantes o las principales empresas que donaron a Obama y Romney. Para averiguar esto, podemos agrupar por nombre de candidato y emplear una variante del método `top` que usamos antes en este capítulo:

```

def get_top_amounts(group, key, n=5):
    totals = group.groupby(key)[“contb_receipt_amt”].sum()
    return totals.nlargest(n)

```

Después agregamos por ocupación y empleador:

```
In [222]: grouped = fec_mrbo.groupby(“cand_nm”)
```

```
In [223]: grouped.apply(get_top_amounts, “contbr_occupation”, n=7)
Out[223]:
```

cand_nm	contbr_occupation	
Obama, Barack	RETIRED	25305116.38
	ATTORNEY	11141982.97

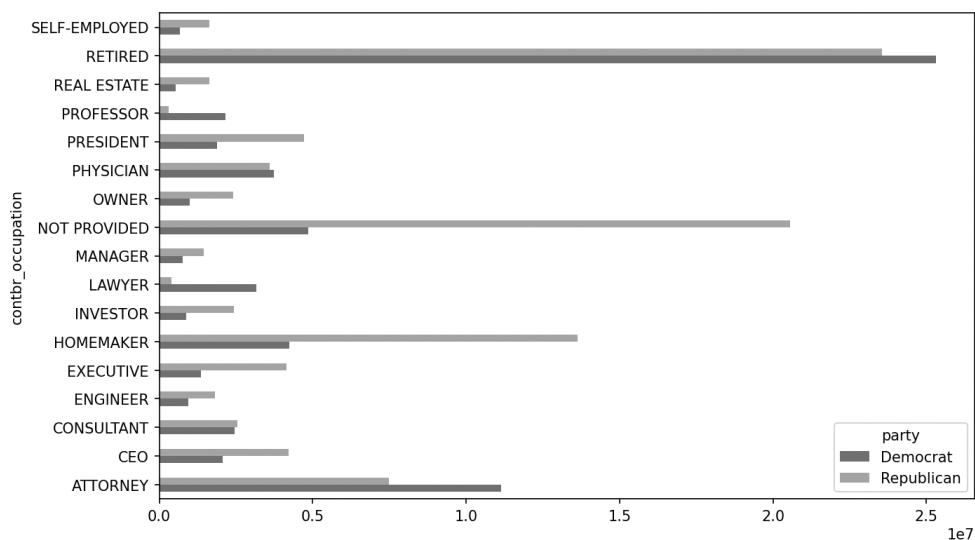


Figura 13.12. Donaciones totales por partido para las principales ocupaciones.

	INFORMATION REQUESTED	4866973.96
		4248875.80
Romney, Mitt	HOMEMAKER	3735124.94
	PHYSICIAN	3160478.87
	LAWYER	2459912.71
	CONSULTANT	11508473.59
	RETIRED	11396894.84
	INFORMATION REQUESTED PER BEST EFFORTS	8147446.22
	HOMEMAKER	5364718.82
	ATTORNEY	2491244.89
	PRESIDENT	2300947.03
	EXECUTIVE	1968386.11
	C.E.O.	

Name: contb_receipt_amt, dtype: float64

```
In [224]: grouped.apply(get_top_amounts, "contbr_employer", n=10)
Out[224]:
```

cand_nm	contbr_employer	
Obama, Barack	RETIRED	22694358.85
	SELF-EMPLOYED	17080985.96
	NOT EMPLOYED	8586308.70
	INFORMATION REQUESTED	5053480.37

	HOMEMAKER	2605408.54
	SELF	1076531.20
	SELF EMPLOYED	469290.00
	STUDENT	318831.45
	VOLUNTEER	257104.00
	MICROSOFT	215585.36
Romney, Mitt	INFORMATION REQUESTED PER BEST EFFORTS	12059527.24
	RETIRED	11506225.71
	HOMEMAKER	8147196.22
	SELF-EMPLOYED	7409860.98
	STUDENT	496490.94
	CREDIT SUISSE	281150.00
	MORGAN STANLEY	267266.00
	GOLDMAN SACH CO.	238250.00
	BARCLAYS CAPITAL	162750.00
	H.I.G. CAPITAL	139500.00

Name: contb_receipt_amt, dtype: float64

Incluir donaciones en contenedores

Una forma útil de analizar estos datos es emplear la función `cut` para discretizar las cantidades donadas por los contribuyentes en contenedores por tamaño de la contribución:

```
In [225]: bins = np.array([0, 1, 10, 100, 1000, 10000,
.....:           100_000, 1_000_000, 10_000_000])
```

```
In [226]: labels = pd.cut(fec_mrbo["contb_receipt_amt"], bins)
```

```
In [227]: labels
Out[227]:
```

411	(10, 100]
412	(100, 1000]
413	(100, 1000]
414	(10, 100]
415	(10, 100]
	...
701381	(10, 100]
701382	(100, 1000]
701383	(1, 10]
701384	(10, 100]
701385	(100, 1000]

Name: contb_receipt_amt, Length: 694282, dtype: category
Categories (8, interval[int64, right]): [(0, 1] < (1, 10] < (10, 100] <

```
(100, 1000] <
(1000, 10000] < (10000, 100000] < (100000, 1000000] <
(1000000, 10000000)]]
```

Podemos ahora agrupar los datos para Obama y Romney por nombre y etiqueta de contenedor para obtener un histograma por tamaño de donación:

```
In [228]: grouped = fec_mrbo.groupby(["cand_nm", "labels"])
```

```
In [229]: grouped.size().unstack(level=0)
Out[229]:
```

cand_nm	Obama, Barack	Romney, Mitt
contb_receipt_amt		
(0, 1]	493	77
(1,10]	40070	3681
(10,100]	372280	31853
(100,1000]	153991	43357
(1000,10000]	22284	26186
(10000,100000]	2	1
(100000,1000000]	3	0
(1000000,10000000]	4	0

Estos datos muestran que Obama recibió un número significativamente mayor de pequeñas donaciones que Romney. Podemos sumar también las cantidades de las contribuciones y normalizar dentro de contenedores para visualizar el porcentaje de donaciones totales de cada tamaño por candidato (la figura 13.13 muestra el gráfico resultante):

```
In [231]: bucket_sums = grouped["contb_receipt_amt"].sum().unstack(level=0)
```

```
In [232]: normed_sums = bucket_sums.div(bucket_sums.sum(axis="columns"),
.....:                                     axis="index")
```

```
In [233]: normed_sums
Out[233]:
```

cand_nm	Obama, Barack	Romney, Mitt
contb_receipt_amt		
(0,1]	0.805182	0.194818
(1,10]	0.918767	0.081233
(10,100]	0.910769	0.089231
(100,1000]	0.710176	0.289824
(1000,10000]	0.447326	0.552674
(10000,100000]	0.823120	0.176880
(100000,1000000]	1.000000	0.000000

```
(10000000, 100000000] 1.000000 0.000000
```

```
In [234]: normed_sums[:-2].plot(kind="barh")
```

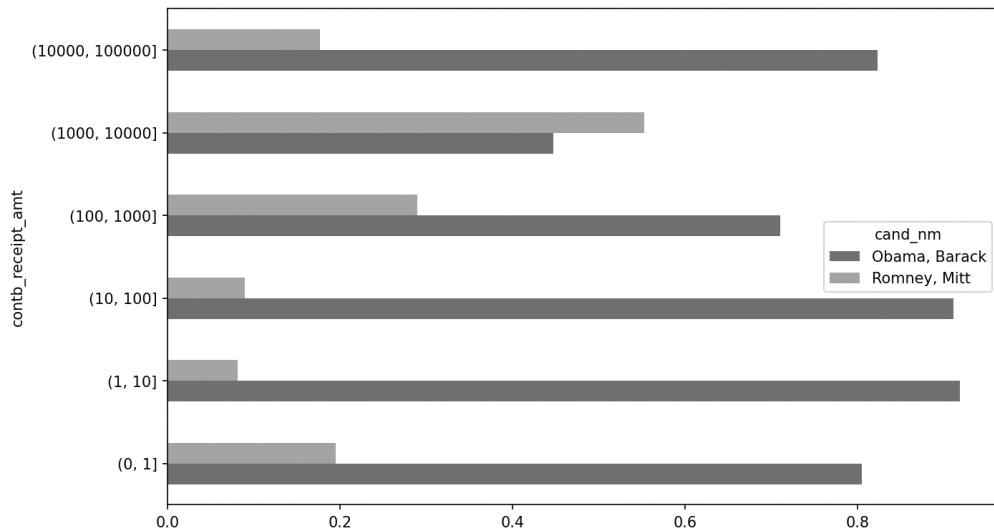


Figura 13.13. Porcentaje de donaciones totales recibidas por los candidatos por cada tamaño de donación.

He excluido los dos contenedores más grandes, porque no son donaciones realizadas por individuos.

Este análisis puede refinarse y mejorarse de varias maneras. Por ejemplo, se podrían agregar donaciones por nombre y código postal del donante para afinar a los donantes que dieron muchas cantidades pequeñas frente a quienes dieron una o dos donaciones grandes. Animo a los lectores a explorar los datos por sí mismos.

Estadísticas de donación por estado

Podemos empezar agregando los datos por candidato y estado:

```
In [235]: grouped = fec_mrbo.groupby(["cand_nm", "contbr_st"])
```

```
In [236]: totals = grouped["contb_receipt_amt"].sum().unstack(level=0).fillna(0)
```

```
In [237]: totals = totals[totals.sum(axis="columns") > 100000]
```

```
In [238]: totals.head(10)
```

```
Out[238]:
```

cand_nm	Obama, Barack	Romney, Mitt
contbr_st		
AK	281840.15	86204.24
AL	543123.48	527303.51
AR	359247.28	105556.00

AZ	1506476.98	1888436.23
CA	23824984.24	11237636.60
CO	2132429.49	1506714.12
CT	2068291.26	3499475.45
DC	4373538.80	1025137.50
DE	336669.14	82712.00
FL	7318178.58	8338458.81

Si dividimos cada fila por la cantidad contribuida total, obtenemos el porcentaje relativo de donaciones totales por estado para cada candidato:

```
In [239]: percent = totals.div(totals.sum(axis="columns"), axis="index")
```

```
In [240]: percent.head(10)
```

```
Out[240]:
```

cand_nm	Obama, Barack	Romney, Mitt
contrb_st		
AK	0.765778	0.234222
AL	0.507390	0.492610
AR	0.772902	0.227098
AZ	0.443745	0.556255
CA	0.679498	0.320502
CO	0.585970	0.414030
CT	0.371476	0.628524
DC	0.810113	0.189887
DE	0.802776	0.197224
FL	0.467417	0.532583

13.6 Conclusión

Hemos llegado al final del libro. En los apéndices he incluido cierto contenido adicional que puede resultar de utilidad.

En los diez años transcurridos desde que se publicó la primera edición de este volumen, Python se ha convertido en un lenguaje popular y de uso general para análisis de datos. Las habilidades de programación desarrolladas aquí seguirán siendo relevantes durante mucho tiempo en el futuro. Espero que las herramientas de programación y las librerías que hemos explorado aquí le sirvan de mucho.

³ Aquí se simplifica suponiendo que Gary Johnson es republicano, aunque después se convirtiera en el candidato del Partido Libertario.

NumPy avanzado

En este apéndice voy a profundizar en la librería NumPy para cálculo de arrays, lo que incluirá más detalles internos sobre el tipo ndarray y otras manipulaciones y algoritmos de arrays más avanzados.

Este apéndice contiene temas varios, por lo tanto, no tiene que leerse necesariamente de forma lineal. A lo largo de sus distintas secciones, generaré datos aleatorios para muchos ejemplos que harán uso del generador de números aleatorios del módulo `numpy.random`:

```
In [11]: rng = np.random.default_rng(seed=12345)
```

A.1 Análisis del objeto ndarray

El ndarray de NumPy ofrece una forma de interpretar un bloque de datos de tipo homogéneo (contiguos o escalonados) como un objeto array multidimensional. El tipo de datos, o `dtype`, determina si se interpretan los datos como punto flotante, entero, booleano o como cualesquiera de los otros tipos que hemos estado viendo.

Parte de lo que hace flexible a ndarray es que cada objeto array es una vista escalonada de un bloque de datos. Quizá algún lector se pregunte, por ejemplo, cómo es que la vista de array `arr[::2, ::-1]` no copia dato alguno. La razón es que el ndarray es algo más que un simple fragmento de memoria y un tipo de datos; también incluye información escalonada, que permite al array moverse por la memoria con distintos tamaños de escalón, paso o incremento. Dicho con más exactitud, el ndarray consiste internamente en lo siguiente:

- Un puntero a los datos, es decir, un bloque de datos de la RAM o de un archivo proyectado en la memoria.
- El tipo de datos o `dtype`, que describe celdas de valor de tamaño fijo del array.
- Una tupla, que indica la forma del array.
- Una tupla de escalones, pasos o incrementos, enteros, que indican el número de bytes que hay que «dar» (como si fueran pasos) para avanzar un elemento a lo largo de una dimensión.

Véase en la figura A.1 un sencillo bosquejo del interior del ndarray.



Figura A.1. El objeto ndarray de NumPy.

Por ejemplo, un array de 10×5 tendría la forma `(10, 5)`:

```
In [12]: np.ones((10, 5)).shape
Out[12]: (10, 5)
```

Un array típico (de orden C) de $3 \times 4 \times 5$ de valores `float64` (8 bytes) tiene los incrementos `(160, 40, 8)` (conocer los escalones o incrementos puede resultar útil porque, en general, cuánto mayores son en un determinado eje, más costoso es realizar cálculos a lo largo de él):

```
In [13]: np.ones((3, 4, 5), dtype=np.float64).strides
Out[13]: (160, 40, 8)
```

Aunque es raro que un usuario habitual de NumPy esté interesado en los pasos de un array, son necesarios para construir vistas de array de «copia cero». Los pasos pueden incluso ser negativos, lo que permite a un array moverse «hacia atrás» en la memoria (este sería el caso, por ejemplo, en un segmento como `obj[::-1]` o `obj[:, ::-1]`).

Jerarquía del tipo de datos NumPy

Es posible que tengamos a veces código que debe comprobar si un array contiene enteros, números de punto flotante, cadenas de texto u objetos Python. Como hay varios tipos de números de punto flotante (desde float16 hasta float128), comprobar que el tipo de datos esté entre una lista de tipos sería muy engorroso. Por suerte, los tipos de datos tienen superclases, como np.integer y np.floating, que se pueden utilizar con la función np.issubdtype:

```
In [14]: ints = np.ones(10, dtype=np.uint16)
In [15]: floats = np.ones(10, dtype=np.float32)
In [16]: np.issubdtype(ints.dtype, np.integer)
Out[16]: True
In [17]: np.issubdtype(floats.dtype, np.floating)
Out[17]: True
```

Podemos ver todas las clases padre de un determinado tipo de datos llamando a su método mro:

```
In [18]: np.float64.mro()
Out[18]:
[numpy.float64,
 numpy.floating,
 numpy.inexact,
 numpy.number,
 numpy.generic,
 float,
object]
```

Por lo tanto, también tenemos:

```
In [19]: np.issubdtype(ints.dtype, np.number)
Out[19]: True
```

La mayoría de los usuarios de NumPy nunca tendrán que saber esto, pero en ocasiones resulta útil. En la figura A.2 podemos ver un diagrama de

la jerarquía de los tipos de datos y las relaciones de subclases padre⁴.

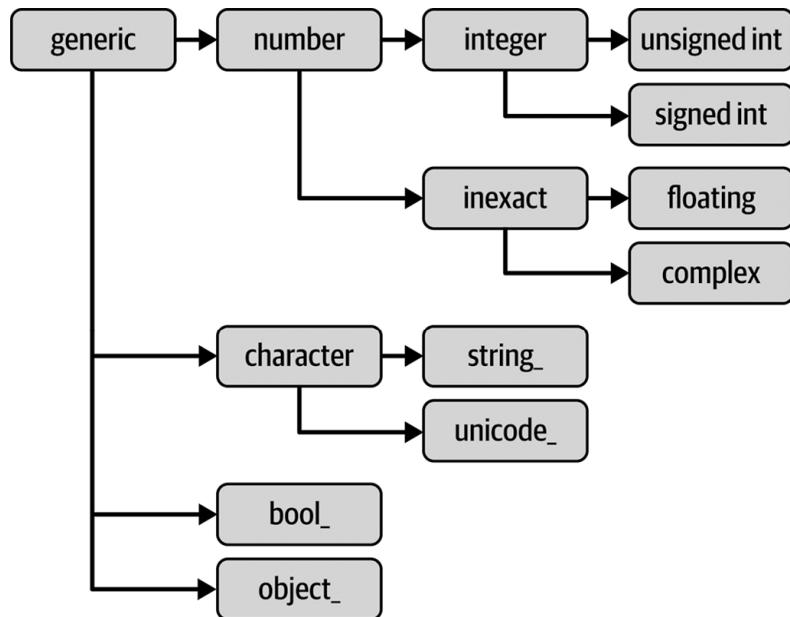


Figura A.2. La jerarquía de clases de los tipos de datos NumPy.

A.2 Manipulación de arrays avanzada

Hay muchas formas de trabajar con arrays más allá del indexado sofisticado, la segmentación y la creación de subconjuntos booleanos. Aunque buena parte de la carga de trabajo de las aplicaciones de análisis de datos es gestionada por funciones de alto nivel en pandas, quizás en algún momento sea necesario escribir un algoritmo de datos que no se encuentre en ninguna de las librerías existentes.

Remodelado de arrays

En muchos casos, se puede convertir un array de una forma a otra sin copiar dato alguno. Para ello, pasamos una tupla que indica la nueva forma al método de instancia de array `reshape`. Por ejemplo, supongamos que tenemos un array unidimensional de valores que deseamos reordenar con forma de matriz (lo que se ilustra en la figura A.3):

```
In [20]: arr = np.arange(8)

In [21]: arr
Out[21]: array([0, 1, 2, 3, 4, 5, 6, 7])

In [22]: arr.reshape((4, 2))
Out[22]:
```

```
array([
       [0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

<code>0</code>	<code>1</code>	<code>2</code>	<code>3</code>	<code>4</code>	<code>5</code>	<code>6</code>	<code>7</code>	<code>8</code>	<code>9</code>	<code>10</code>	<code>11</code>												
<code>arr.reshape((4,3), order=?)</code>																							
Orden de C (principal de fila)																							
Orden de Fortran (principal de columna)																							
<table border="1"> <tbody> <tr> <td><code>0</code></td><td><code>1</code></td><td><code>2</code></td></tr> <tr> <td><code>3</code></td><td><code>4</code></td><td><code>5</code></td></tr> <tr> <td><code>6</code></td><td><code>7</code></td><td><code>8</code></td></tr> <tr> <td><code>9</code></td><td><code>10</code></td><td><code>11</code></td></tr> </tbody> </table>												<code>0</code>	<code>1</code>	<code>2</code>	<code>3</code>	<code>4</code>	<code>5</code>	<code>6</code>	<code>7</code>	<code>8</code>	<code>9</code>	<code>10</code>	<code>11</code>
<code>0</code>	<code>1</code>	<code>2</code>																					
<code>3</code>	<code>4</code>	<code>5</code>																					
<code>6</code>	<code>7</code>	<code>8</code>																					
<code>9</code>	<code>10</code>	<code>11</code>																					
<code>orden = 'C'</code>																							
<table border="1"> <tbody> <tr> <td><code>0</code></td><td><code>4</code></td><td><code>8</code></td></tr> <tr> <td><code>1</code></td><td><code>5</code></td><td><code>9</code></td></tr> <tr> <td><code>2</code></td><td><code>6</code></td><td><code>10</code></td></tr> <tr> <td><code>3</code></td><td><code>7</code></td><td><code>11</code></td></tr> </tbody> </table>												<code>0</code>	<code>4</code>	<code>8</code>	<code>1</code>	<code>5</code>	<code>9</code>	<code>2</code>	<code>6</code>	<code>10</code>	<code>3</code>	<code>7</code>	<code>11</code>
<code>0</code>	<code>4</code>	<code>8</code>																					
<code>1</code>	<code>5</code>	<code>9</code>																					
<code>2</code>	<code>6</code>	<code>10</code>																					
<code>3</code>	<code>7</code>	<code>11</code>																					
<code>orden = 'F'</code>																							

Figura A.3. Remodelación en orden de C (principal de fila) o de FORTRAN (principal de columna).

También se puede cambiar la forma de un array multidimensional:

```
In [23]: arr.reshape((4, 2)).reshape((2, 4))
Out[23]:
```

```
array([
       [0, 1, 2, 3],
       [4, 5, 6, 7]])
```

Una de las dimensiones de forma pasadas puede ser -1 , en cuyo caso el valor utilizado para dicha dimensión se deducirá de los datos:

```
In [24]: arr = np.arange(15)
```

```
In [25]: arr.reshape((5, -1))
```

```
Out[25]:
```

```
array([
        [ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [ 9, 10, 11],
        [12, 13, 14]])
```

Como el atributo `shape` de un array es una tupla, también se le puede pasar a `reshape`:

```
In [26]: other_arr = np.ones((3, 5))
```

```
In [27]: other_arr.shape
```

```
Out[27]: (3, 5)
```

```
In [28]: arr.reshape(other_arr.shape)
```

```
Out[28]:
```

```
array([
        [ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14]])
```

La operación contraria a `reshape` de una sola dimensión a muchas se conoce normalmente como aplanado (`flatten` o `ravel`):

```
In [29]: arr = np.arange(15).reshape((5, 3))
```

```
In [30]: arr
```

```
Out[30]:
```

```
array([
        [ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
```

```
[ 9, 10, 11],  
[12, 13, 14]])
```

```
In [31]: arr.ravel()  
Out[31]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,  
13, 14])
```

La operación `ravel` no produce una copia de los valores subyacentes si los valores del resultado eran contiguos en el array original.

El método `flatten` se comporta igual que `ravel`, excepto que siempre devuelve una copia de los datos:

```
In [32]: arr.flatten()  
Out[32]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,  
13, 14])
```

Los datos se pueden remodelar o aplanar en distintos órdenes. Este es un tema con ligeros matices para los nuevos usuarios de NumPy, por lo tanto es el siguiente apartado.

Orden de C frente a FORTRAN

NumPy es capaz de adaptarse a muchas disposiciones diferentes de los datos en la memoria. De forma predeterminada, los arrays NumPy se crean en orden principal de fila. Espacialmente, esto significa que si tenemos un array de datos bidimensional, los elementos de cada fila del array se almacenan en posiciones adyacentes de la memoria. La alternativa a este tipo de orden es el orden principal de columna, donde son los valores de cada columna de datos los que se almacenan en posiciones de memoria adyacentes.

Por razones históricas, el orden principal de fila y columna se conocen también como orden de C y de FORTRAN, respectivamente. En el lenguaje FORTRAN 77, las matrices están todas en orden principal de columna.

Funciones como `reshape` y `ravel` aceptan un argumento `order` que indica el orden en el que se deben utilizar los datos del array. Normalmente esto está fijado en '`C`' o '`F`' en la mayoría de los casos (también existen las

opciones menos usadas ‘A’ y ‘K’; en la documentación de NumPy se dispone de más información, y en la figura A.3 anterior podemos ver una ilustración de estas opciones):

```
In [33]: arr = np.arange(12).reshape((3, 4))
```

```
In [34]: arr
```

```
Out[34]:
```

```
array([
       [ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
In [35]: arr.ravel()
```

```
Out[35]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
In [36]: arr.ravel('F')
```

```
Out[36]: array([ 0,  4,  8,  1,  5,  9,  2,  6, 10,  3,  7, 11])
```

Remodelar arrays con más de dos dimensiones puede ser un poco complicado (véase la figura A.3). La diferencia esencial entre el orden de C y FORTRAN es el modo en el que se recorren las dimensiones:

- C/orden principal de fila: Recorre las dimensiones superiores en primer lugar (por ejemplo, eje 1 antes de avanzar al eje 0).
- FORTRAN/ orden principal de columna: Recorre las dimensiones superiores en último lugar (por ejemplo, eje 0 antes de avanzar al eje 1).

Concatenación y división de arrays

La función `numpy.concatenate` toma una secuencia (tupla, lista, etc.) de arrays y los une en orden a lo largo del eje de entrada:

```
In [37]: arr1 = np.array([[1, 2, 3], [4, 5, 6]])
```

```
In [38]: arr2 = np.array([[7, 8, 9], [10, 11, 12]])
```

```
In [39]: np.concatenate([arr1, arr2], axis=0)
```

```
Out[39]:
```

```
array([
      [ 1, 2, 3],
      [ 4, 5, 6],
      [ 7, 8, 9],
      [10, 11, 12]])
```

```
In [40]: np.concatenate([arr1, arr2], axis=1)
Out[40]:
```

```
array([[ 1, 2, 3, 7, 8, 9],
       [ 4, 5, 6, 10, 11, 12]])
```

Hay ciertas funciones, como `vstack` y `hstack`, que son cómodas para tipos de concatenación habituales. Las operaciones anteriores se podrían haber expresado así:

```
In [41]: np.vstack((arr1, arr2))
Out[41]:
```

```
array([
      [ 1, 2, 3],
      [ 4, 5, 6],
      [ 7, 8, 9],
      [10, 11, 12]])
```

```
In [42]: np.hstack((arr1, arr2))
Out[42]:
```

```
array([[ 1, 2, 3, 7, 8, 9],
       [ 4, 5, 6, 10, 11, 12]])
```

Por otro lado, `split` segmenta un array en varios a lo largo de un eje:

```
In [43]: arr = rng.standard_normal((5, 2))
```

```
In [44]: arr
Out[44]:
```

```
array([[ -1.4238,  1.2637],
       [-0.8707, -0.2592],
```

```

[-0.0753, -0.7409],
[-1.3678, 0.6489],
[ 0.3611, -1.9529]])

In [45]: first, second, third = np.split(arr, [1, 3])

In [46]: first
Out[46]: array([[-1.4238, 1.2637]])

In [47]: second
Out[47]:

array([
       [-0.8707, -0.2592],
       [-0.0753, -0.7409]]))

In [48]: third
Out[48]:

```



```

array([
       [-1.3678, 0.6489],
       [ 0.3611, -1.9529]])

```

El valor `[1, 3]` pasado a `np.split` indica los índices en los que dividir el array en partes.

La tabla A.1 ofrece una lista de todas las funciones importantes de concatenación y división, algunas de las cuales se incluyen solamente como una utilidad más de la función general `concatenate`.

Tabla A.1. Funciones de concatenación de arrays.

Función	Descripción
<code>concatenate</code>	La función más general, que concatena colecciones de arrays a lo largo de un eje.
<code>vstack</code> , <code>row_stack</code>	Apila arrays por filas (a lo largo del eje 0).
<code>hstack</code>	Apila arrays por columnas (a lo largo del eje 1).
<code>column_stack</code>	Igual que <code>hstack</code> , pero convierte en primer lugar arrays unidimensionales en

	vectores de columna de dos dimensiones.
dstack	Apila arrays por «profundidad» (a lo largo del eje 2).
split	Divide el array en las posiciones pasadas a lo largo de un determinado eje.
hsplit/vsplit	Funciones de utilidad para segmentar en el eje 0 y 1, respectivamente.

Auxiliares de apilamiento: r_ y c_

Hay dos objetos especiales en el espacio de nombres NumPy, r_ y c_, que abrevian el apilamiento de arrays:

```
In [49]: arr = np.arange(6)
```

```
In [50]: arr1 = arr.reshape((3, 2))
```

```
In [51]: arr2 = rng.standard_normal((3, 2))
```

```
In [52]: np.r_[arr1, arr2]
```

```
Out[52]:
```

```
array([
       [ 0. ,  1. ],
       [ 2. ,  3. ],
       [ 4. ,  5. ],
       [ 2.3474,  0.9685],
       [-0.7594,  0.9022],
       [-0.467 , -0.0607]])
```

```
In [53]: np.c_[np.r_[arr1, arr2], arr]
```

```
Out[53]:
```

```
array([
       [ 0. ,  1. ,  0. ],
       [ 2. ,  3. ,  1. ],
       [ 4. ,  5. ,  2. ],
       [ 2.3474,  0.9685,  3. ],
       [-0.7594,  0.9022,  4. ],
       [-0.467 , -0.0607,  5. ]])
```

Estos objetos pueden convertir además segmentos en arrays:

```
In [54]: np.c_[1:6, -10:-5]
Out[54]:
```

```
array([
       [ 1, -10],
       [ 2, -9],
       [ 3, -8],
       [ 4, -7],
       [ 5, -6]])
```

En el docstring se puede obtener más información de lo que se puede hacer con `c_` y `r_`.

Repetición de elementos: `tile` y `repeat`

Dos herramientas útiles para repetir o replicar arrays y producirlos así de mayor tamaño son las funciones `repeat` y `tile`. La primera, `repeat`, replica cada elemento de un array un cierto número de veces, creando uno más grande:

```
In [55]: arr = np.arange(3)

In [56]: arr
Out[56]: array([0, 1, 2])

In [57]: arr.repeat(3)
Out[57]: array([0, 0, 0, 1, 1, 1, 2, 2, 2])
```



La necesidad de replicar o repetir arrays puede ser menos habitual con NumPy que con otros marcos de programación de arrays como MATLAB. Una razón de esto es que la difusión, el tema de nuestra siguiente sección, suele cubrir mejor esta necesidad.

De forma predeterminada, si se pasa un entero, cada elemento se repetirá ese número de veces. Si se pasa un array de enteros, cada elemento puede repetirse un número distinto de veces:

```
In [58]: arr.repeat([2, 3, 4])
Out[58]: array([0, 0, 1, 1, 1, 2, 2, 2, 2])
```

En los arrays multidimensionales, sus elementos pueden repetirse a lo largo de un determinado eje:

```
In [59]: arr = rng.standard_normal((2, 2))
```

```
In [60]: arr
```

```
Out[60]:
```

```
array([ [ 0.7888, -1.2567],  
       [ 0.5759,  1.399 ]])
```

```
In [61]: arr.repeat(2, axis=0)
```

```
Out[61]:
```

```
array([ [ 0.7888, -1.2567],  
       [ 0.7888, -1.2567],  
       [ 0.5759,  1.399 ],  
       [ 0.5759,  1.399 ]])
```

Conviene observar que si no se pasa ningún eje, en primer lugar el array será aplanado, que probablemente no es lo que queremos. Igualmente podemos pasar un array de enteros al repetir un array multidimensional para (de nuevo) repetir un determinado fragmento un número diferente de veces:

```
In [62]: arr.repeat([2, 3], axis=0)
```

```
Out[62]:
```

```
array([ [ 0.7888, -1.2567],  
       [ 0.7888, -1.2567],  
       [ 0.5759,  1.399 ],  
       [ 0.5759,  1.399 ],  
       [ 0.5759,  1.399 ]])
```

```
In [63]: arr.repeat([2, 3], axis=1)
```

```
Out[63]:
```

```
array([ [ 0.7888,  0.7888, -1.2567, -1.2567, -1.2567],
```

```
[ 0.5759, 0.5759, 1.399 , 1.399 , 1.399 ]])
```

Por otro lado, `tile` es un atajo para apilar copias de un array a lo largo de un eje. Se puede pensar en ello de manera visual como algo parecido a «colocar baldosas»:

```
In [64]: arr  
Out[64]:
```

```
array([ 0.7888, -1.2567],  
      [ 0.5759, 1.399 ]))
```

```
In [65]: np.tile(arr, 2)  
Out[65]:
```

```
array([ 0.7888, -1.2567, 0.7888, -1.2567],  
      [ 0.5759, 1.399 , 0.5759, 1.399 ]))
```

El segundo argumento es el numero de baldosas; con un escalar, la colocación de baldosas se hace fila a fila, en lugar de columna a columna. El segundo argumento de `tile` puede ser una tupla que indica la disposición del «mosaico»:

```
In [66]: arr  
Out[66]:
```

```
array([ 0.7888, -1.2567],  
      [ 0.5759, 1.399 ]))
```

```
In [67]: np.tile(arr, (2, 1))  
Out[67]:
```

```
array([ 0.7888, -1.2567],  
      [ 0.5759, 1.399 ],  
      [ 0.7888, -1.2567],  
      [ 0.5759, 1.399 ]))
```

```
[ 0.5759, 1.399 ]])
```

```
In [68]: np.tile(arr, (3, 2))  
Out[68]:
```

```
array([ 0.7888, -1.2567, 0.7888, -1.2567],  
      [ 0.5759, 1.399 , 0.5759, 1.399 ],  
      [ 0.7888, -1.2567, 0.7888, -1.2567],  
      [ 0.5759, 1.399 , 0.5759, 1.399 ],  
      [ 0.7888, -1.2567, 0.7888, -1.2567],  
      [ 0.5759, 1.399 , 0.5759, 1.399 ]))
```

Equivalentes del indexado sofisticado: take y put

Como quizá mis lectores recuerden del capítulo 4, una forma de obtener subconjuntos de arrays es mediante el indexado sofisticado y utilizando arrays de enteros:

```
In [69]: arr = np.arange(10) * 100  
In [70]: inds = [7, 1, 2, 6]  
In [71]: arr[inds]  
Out[71]: array([700, 100, 200, 600])
```

Existen métodos ndarray alternativos que son útiles en el caso especial de hacer una selección solamente en un único eje:

```
In [72]: arr.take(inds)  
Out[72]: array([700, 100, 200, 600])  
  
In [73]: arr.put(inds, 42)  
  
In [74]: arr  
Out[74]: array([ 0, 42, 42, 300, 400, 500, 42, 42, 800,  
 900])  
  
In [75]: arr.put(inds, [40, 41, 42, 43])  
  
In [76]: arr  
Out[76]: array([ 0, 41, 42, 300, 400, 500, 43, 40, 800,  
 900])
```

Para utilizar `take` a lo largo de otros ejes, se puede pasar la palabra clave `axis`:

```
In [77]: inds = [2, 0, 2, 1]  
In [78]: arr = rng.standard_normal((2, 4))
```

```
In [79]: arr  
Out[79]:
```

```
array([[ 1.3223, -0.2997,  0.9029, -1.6216],  
       [-0.1582,  0.4495, -1.3436, -0.0817]])
```

```
In [80]: arr.take(inds, axis=1)  
Out[80]:  
  
array([[ 0.9029,  1.3223,  0.9029, -0.2997],  
       [-1.3436, -0.1582, -1.3436,  0.4495]])
```

La función `put` no acepta un argumento `axis`; lo que hace realmente es indexar en la versión aplanada (unidimensional, en orden de C) del array. De este modo, cuando es necesario establecer elementos usando un array de índice en otros ejes, es mejor emplear la indexación basada en corchetes ([]).

A.3 Difusión

La difusión controla el modo en que funcionan las operaciones entre arrays de distintas formas. Puede ser una función de gran potencia, pero puede también causar confusión, incluso a los usuarios más expertos. El ejemplo más sencillo de difusión se produce al combinar un valor escalar con un array:

```
In [81]: arr = np.arange(5)  
In [82]: arr  
Out[82]: array([0, 1, 2, 3, 4])
```

```
In [83]: arr * 4  
Out[83]: array([ 0, 4, 8, 12, 16])
```

Aquí decimos que el valor escalar 4 ha sido difundido al resto de los elementos de la operación de multiplicación.

Por ejemplo, podemos degradar cada columna de un array restando las medias de las columnas. En este caso, solo es necesario restar un array que contenga la media de cada columna:

```
In [84]: arr = rng.standard_normal((4, 3))
```

```
In [85]: arr.mean(0)  
Out[85]: array([0.1206, 0.243 , 0.1444])
```

```
In [86]: demeaned = arr - arr.mean(0)
```

```
In [87]: demeaned  
Out[87]:
```

```
array([ [ 1.6042, 2.3751, 0.633 ],  
       [ 0.7081, -1.202 , -1.3538],  
       [-1.5329, 0.2985, 0.6076],  
       [-0.7793, -1.4717, 0.1132]])
```

```
In [88]: demeaned.mean(0)  
Out[88]: array([ 0., -0., 0.])
```

La figura A.4 muestra una ilustración de esta operación. Degrado las filas como operación de difusión requiere tener un poco más de cuidado. Por suerte, es posible difundir valores que pueden ser de pocas dimensiones a lo largo de cualquier dimensión de un array (al igual que restar la media de la fila de cada columna de un array bidimensional) siempre que se sigan las reglas. Esto nos lleva a la regla de la difusión.

La regla de la difusión

Dos arrays son compatibles para difusión si para cada dimensión final (es decir, empezando desde el final) las longitudes de los ejes coinciden o si

cualquiera de las longitudes es 1. Se produce entonces la difusión a lo largo de las dimensiones ausentes o de longitud 1.

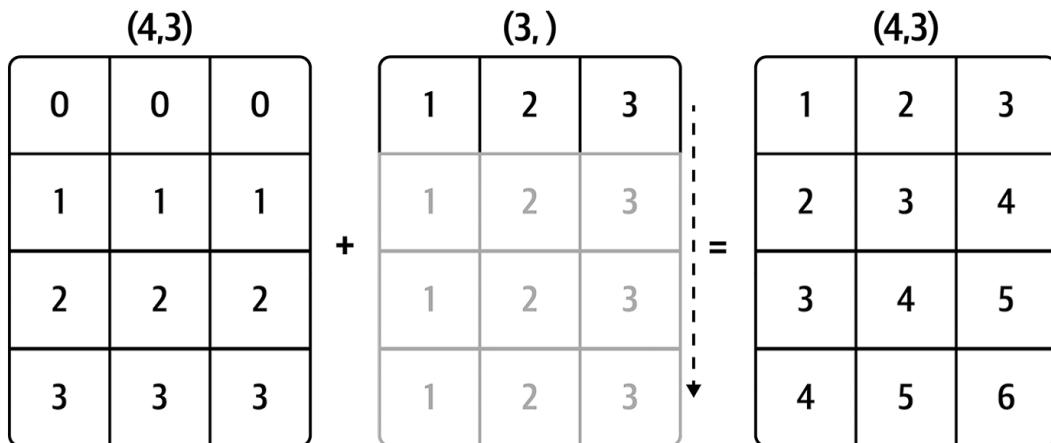


Figura A.4. Difusión a lo largo del eje 0 con un array unidimensional.

Aun siendo un usuario experto de NumPy, con frecuencia me descubro teniendo que parar y dibujar un diagrama cuando pienso en la regla de la difusión. Veamos el último ejemplo y supongamos que lo que queremos hacer es restar el valor medio de cada fila. Como `arr.mean(0)` tiene longitud 3, es compatible para difusión a lo largo del eje 0 porque la dimensión final de `arr` es 3, y por lo tanto, coincide. Según las reglas, para restar a lo largo del eje 1 (es decir, restar la media de la fila de cada fila), el array más pequeño debe tener la forma $(4, 1)$:

```
In [89]: arr
Out[89]:
```

```
array([
    [ 1.7247,  2.6182,  0.7774],
    [ 0.8286, -0.959 , -1.2094],
    [-1.4123,  0.5415,  0.7519],
    [-0.6588, -1.2287,  0.2576]])
```

```
In [90]: row_means = arr.mean(1)
```

```
In [91]: row_means.shape
Out[91]: (4, )
```

```
In [92]: row_means.reshape((4, 1))  
Out[92]:
```

```
array([ 1.7068],  
[-0.4466],  
[-0.0396],  
[-0.5433]))
```

```
In [93]: demeaned = arr - row_means.reshape((4, 1))
```

```
In [94]: demeaned.mean(1)  
Out[94]: array([-0., 0., 0., 0.])
```

La figura A.5 ilustra esta operación.

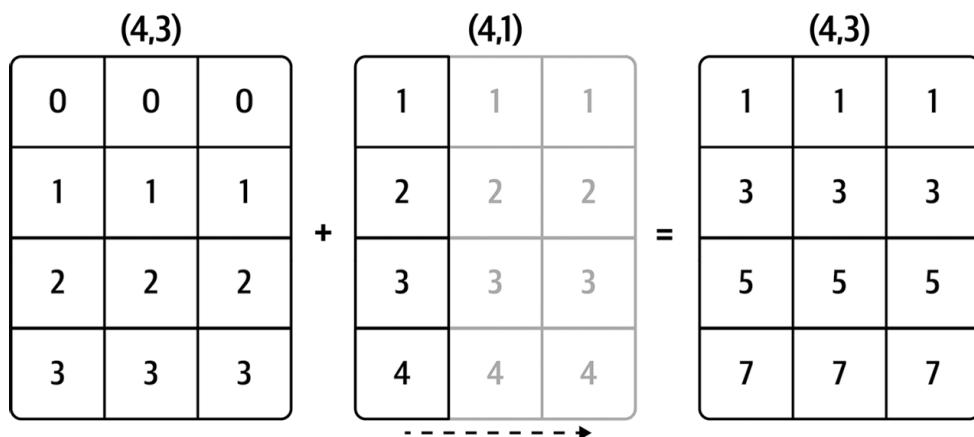


Figura A.5. Difusión a lo largo del eje 1 de un array bidimensional.

En la figura A.6 podemos ver otra ilustración, tras añadir en esta ocasión un array bidimensional a otro tridimensional a lo largo del eje

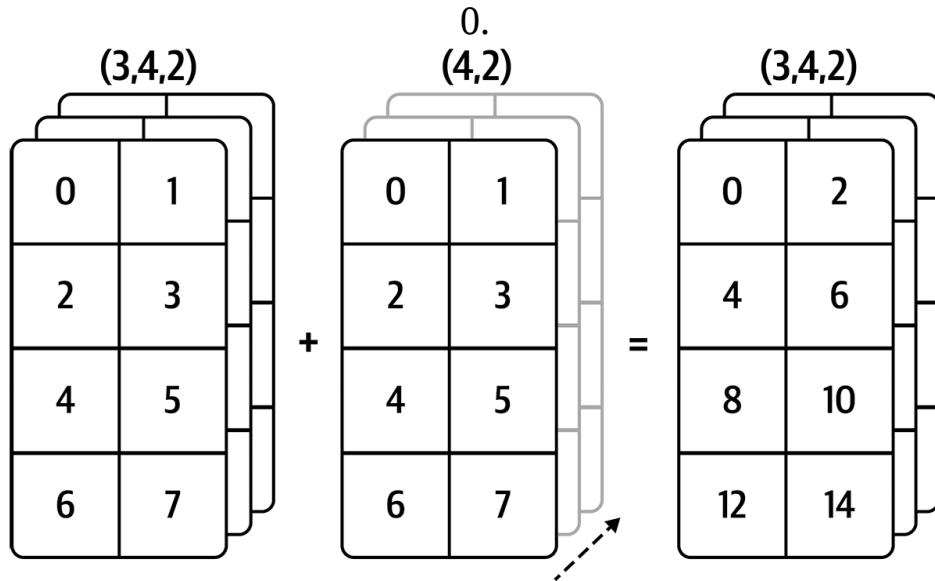


Figura A.6. Difusión a lo largo del eje 0 de un array tridimensional.

Difusión a lo largo de otros ejes

Difundir con arrays de muchas dimensiones puede parecer bastante complicado, pero en realidad es cuestión de seguir las reglas. Si no se siguen, obtenemos un error como este:

```
In [95]: arr-arr.mean(1)
```

```
ValueError      Traceback (most recent call last)
```

```
<ipython-input-95-8b8ada26fac0> in <module>
    1 arr-arr.mean(1)
```

```
ValueError: operands could not be broadcast together with shapes (4, 3) (4, )
```

Es bastante habitual querer realizar una operación aritmética con un array de pocas dimensiones a lo largo de ejes que no sean el eje 0. Según la regla de la difusión, las «dimensiones de difusión» deben ser 1 en el array más pequeño. En el ejemplo de degradado de fila mostrado aquí, esto significa remodelar la fila para que tenga la forma (4, 1) en lugar de (4,):

```
In [96]: arr-arr.mean(1).reshape((4, 1))  
Out[96]:
```

```
array([ [ 0.018 , 0.9114, -0.9294],  
       [ 1.2752, -0.5124, -0.7628],  
       [-1.3727, 0.5811, 0.7915],  
       [-0.1155, -0.6854, 0.8009]])
```

En el caso de tener tres dimensiones, difundir a lo largo de cualquiera de las tres es simplemente cuestión de remodelar los datos para que sean compatibles con la forma. La figura A.7 permite visualizar las formas requeridas para difundir a lo largo de cada eje de un array tridimensional.

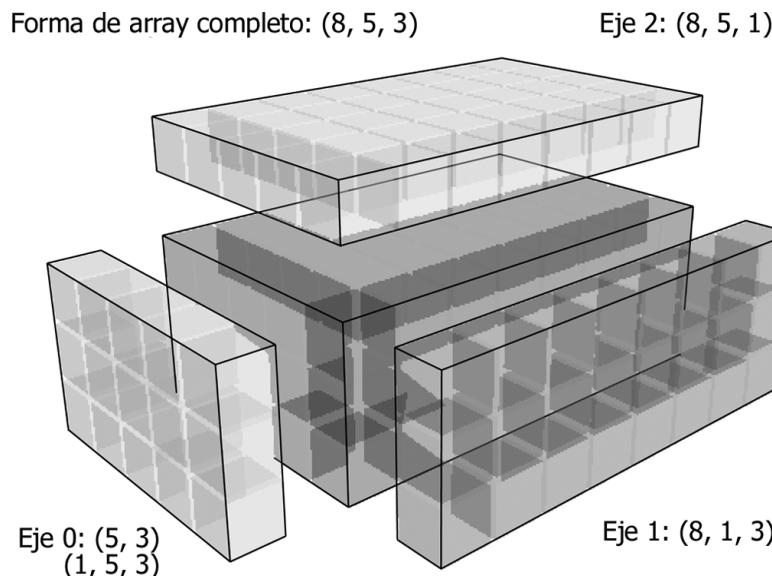


Figura A.7. Formas de array bidimensional compatibles para su difusión a lo largo de un array tridimensional.

Por tanto, un problema habitual es tener la necesidad de añadir un nuevo eje con longitud 1 específicamente para la difusión. Utilizar `reshape` es una opción, pero insertar un eje requiere construir una tupla indicando la nueva forma, y a menudo esto puede resultar muy tedioso. De ahí que los arrays NumPy ofrezcan una sintaxis especial para insertar nuevos ejes indexando. Utilizamos el atributo especial `np.newaxis` junto con segmentos «completos» para insertar el nuevo eje:

```
In [97]: arr = np.zeros((4, 4))

In [98]: arr_3d = arr[:, np.newaxis, :]

In [99]: arr_3d.shape
Out[99]: (4, 1, 4)

In [100]: arr_1d = rng.standard_normal(3)

In [101]: arr_1d[:, np.newaxis]
Out[101]:

array([[ 0.3129,
       -0.1308],
      [ 1.27 ]])

In [102]: arr_1d[np.newaxis, :]
Out[102]: array([[ 0.3129, -0.1308,  1.27 ]])
```

Así, si tenemos un array tridimensional y queremos degradar el eje 2, tendríamos que escribir:

```
In [103]: arr = rng.standard_normal((3, 4, 5))

In [104]: depth_means = arr.mean(2)

In [105]: depth_means
Out[105]:

array([      [ 0.0431,  0.2747, -0.1885, -0.2014],
        [-0.5732, -0.5467,  0.1183, -0.6301],
        [ 0.0972,  0.5954,  0.0331, -0.6002]])


In [106]: depth_means.shape
Out[106]: (3, 4)

In [107]: demeaned = arr-depth_means[:, :, np.newaxis]

In [108]: demeaned.mean(2)
Out[108]:


array([      [ 0., -0.,  0., -0.],
        [ 0., -0., -0., -0.],
```

```
[ 0., 0., 0., 0.]])
```

Es posible que algún lector se pregunte si hay una forma de generalizar la degradación a lo largo de un eje sin sacrificar el rendimiento. La hay, pero requiere algunos ejercicios de indexación:

```
def demean_axis(arr, axis=0):
    means = arr.mean(axis)

    # Esto generaliza cosas como [:, :, np.newaxis] a N
    # dimensiones
    indexer = [slice(None)] * arr.ndim

    indexer[axis] = np.newaxis
    return arr-means[indexer]
```

Configuración de valores de array por difusión

La misma regla de la difusión que rige las operaciones aritméticas aplica a la configuración de valores mediante indexado de arrays. En un caso sencillo podemos hacer cosas como esta:

```
In [109]: arr = np.zeros((4, 3))
```

```
In [110]: arr[:] = 5
```

```
In [111]: arr
Out[111]:
```

```
array([
        [5., 5., 5.],
        [5., 5., 5.],
        [5., 5., 5.],
        [5., 5., 5.]])
```

Sin embargo, si tenemos un array unidimensional de valores que queremos organizar como las columnas del array, podemos hacerlo siempre que la forma sea compatible:

```
In [112]: col = np.array([1.28, -0.42, 0.44, 1.6])
```

```
In [113]: arr[:, np.newaxis]  
In [114]: arr  
Out[114]:  
  
array([ [ 1.28, 1.28, 1.28],  
       [-0.42, -0.42, -0.42],  
       [ 0.44, 0.44, 0.44],  
       [ 1.6 , 1.6 , 1.6 ]])  
  
In [115]: arr[:2] = [[-1.37], [0.509]]  
In [116]: arr  
Out[116]:  
  
array([ [-1.37 , -1.37 , -1.37 ],  
       [ 0.509, 0.509, 0.509],  
       [ 0.44 , 0.44 , 0.44 ],  
       [ 1.6 , 1.6 , 1.6 ]])
```

A.4 Uso avanzado de ufuncs

Aunque muchos usuarios de NumPy solo utilizarán las rápidas operaciones elemento a elemento que ofrecen las funciones universales, hay una serie de características adicionales que pueden ayudar a escribir código más conciso sin bucles explícitos.

Métodos de instancia ufunc

Cada una de las ufuncs binarias de NumPy dispone de métodos especiales para realizar ciertos tipos de operaciones especiales vectorizadas. Se resumen en la tabla A.2, pero voy a dar algunos ejemplos concretos para ilustrar su funcionamiento.

La operación `reduce` toma un único array y agrega sus valores (a lo largo de un eje de manera opcional), realizando una secuencia de

operaciones binarias. Por ejemplo, una forma alternativa de sumar elementos en un array es utilizando `np.add.reduce`:

```
In [117]: arr = np.arange(10)
```

```
In [118]: np.add.reduce(arr)  
Out[118]: 45
```

```
In [119]: arr.sum()  
Out[119]: 45
```

El valor inicial (por ejemplo 0 para `add`) depende de la función universal. Si se pasa un eje, la reducción se lleva a cabo a lo largo de dicho eje, lo que permite responder brevemente a ciertos tipos de preguntas. Como ejemplo menos mundano, podemos utilizar `np.logical_and` para comprobar si los valores de cada fila de un array están ordenados:

```
In [120]: my_rng = np.random.default_rng(12346) # por  
reproducibilidad
```

```
In [121]: arr = my_rng.standard_normal((5, 5))
```

```
In [122]: arr  
Out[122]:
```

```
array([[ -0.9039,  0.1571,  0.8976, -0.7622, -0.1763],  
       [ 0.053 , -1.6284, -0.1775,  1.9636,  1.7813],  
       [-0.8797, -1.6985, -1.8189,  0.119 , -0.4441],  
       [ 0.7691, -0.0343,  0.3925,  0.7589, -0.0705],  
       [ 1.0498,  1.0297, -0.4201,  0.7863,  0.9612]])
```

```
In [123]: arr[::-2].sort(1) # ordena algunas filas
```

```
In [124]: arr[:, :-1] < arr[:, 1:]  
Out[124]:
```

```
array([[ True,  True,  True,  True],  
       [False,  True,  True, False],  
       [ True,  True,  True,  True],  
       [False,  True,  True, False],  
       [ True,  True,  True,  True]])
```

```
In [125]: np.logical_and.reduce(arr[:, :-1] < arr[:, 1:],  
axis=1)  
Out[125]: array([ True, False, True, False, True])
```

Tengamos en cuenta que `logical_and.reduce` es equivalente al método `all`.

El método `ufunc accumulate` está asociado a `reduce`, igual que `cumsum` lo está a `sum`. Produce un array del mismo tamaño con los valores «acumulados» intermedios:

```
In [126]: arr = np.arange(15).reshape((3, 5))
```

```
In [127]: np.add.accumulate(arr, axis=1)  
Out[127]:
```

```
array([ [ 0,  1,  3,  6, 10],  
       [ 5, 11, 18, 26, 35],  
       [10, 21, 33, 46, 60]])
```

`outer` realiza un producto vectorial por pares entre dos arrays:

```
In [128]: arr = np.arange(3).repeat([1, 2, 2])
```

```
In [129]: arr  
Out[129]: array([0, 1, 1, 2, 2])
```

```
In [130]: np.multiply.outer(arr, np.arange(5))  
Out[130]:
```

```
array([ [0,  0,  0,  0,  0],  
       [0,  1,  2,  3,  4],  
       [0,  1,  2,  3,  4],  
       [0,  2,  4,  6,  8],  
       [0,  2,  4,  6,  8]])
```

El resultado de `outer` tendrá una dimensión que es la concatenación de las dimensiones de las entradas:

```
In [131]: x, y = rng.standard_normal((3, 4)),  
rng.standard_normal(5)  
  
In [132]: result = np.subtract.outer(x, y)  
  
In [133]: result.shape  
Out[133]: (3, 4, 5)
```

El último método, `reduceat`, realiza una «reducción local», básicamente una operación «de agrupamiento» de array en la que fragmentos del array se agregan juntos. Acepta una secuencia de «bordes de contenedor» que indica cómo dividir y agregar los valores:

```
In [134]: arr = np.arange(10)  
  
In [135]: np.add.reduceat(arr, [0, 5, 8])  
Out[135]: array([10, 18, 17])
```

Los resultados son las reducciones (en este caso, sumas) realizadas sobre `arr[0:5]`, `arr[5:8]` y `arr[8:]`. Como con los otros métodos, se puede pasar un argumento `axis`:

```
In [136]: arr = np.multiply.outer(np.arange(4),  
np.arange(5))  
  
In [137]: arr  
Out[137]:  
  
array([ [ 0,  0,  0,  0,  0],  
       [ 0,  1,  2,  3,  4],  
       [ 0,  2,  4,  6,  8],  
       [ 0,  3,  6,  9, 12]])  
  
In [138]: np.add.reduceat(arr, [0, 2, 4], axis=1)  
Out[138]:  
  
array([ [ 0,  0,  0],  
       [ 1,  5,  4],  
       [ 2, 10,  8],  
       [ 3, 15, 12]])
```

En la tabla A.2 se puede encontrar un listado parcial de los métodos ufunc.

Tabla A.2. Métodos ufunc.

Método	Descripción
accumulate(x)	Agrega valores, conservando todos los agregados parciales.
at(x, indices, b=None)	Realiza una operación sobre x en los índices especificados. El argumento b es la segunda entrada a ufuncs que requiere dos entradas de array.
reduce(x)	Agrega valores por aplicaciones sucesivas de la operación.
reduceat(x, bins)	Reducción «local» o de «agrupamiento»; reduce segmentos contiguos de datos para producir un array agregado.
outer(x, y)	Aplica la operación a todos los pares de elementos de x e y; el array resultante tiene la forma x.shape + y.shape.

Escribir nuevas ufuncs en Python

Hay varias formas de crear ufuncs de NumPy personalizadas. La más general es utilizando la API C de NumPy, pero esta opción queda más allá del alcance de este libro. En esta sección veremos las ufuncs puras de Python.

`numpy.frompyfunc` acepta una función de Python junto con una especificación para el número de entradas y salidas. Por ejemplo, una función sencilla que sume elemento a elemento se especificaría como:

In [139]: `def add_elements(x, y):`

```
.....:             return x + y
```

In [140]: `addThem = np.frompyfunc(add_elements, 2, 1)`

In [141]: `addThem(np.arange(8), np.arange(8))`
Out[141]: `array([0, 2, 4, 6, 8, 10, 12, 14], dtype=object)`

Las funciones creadas utilizando `frompyfunc` siempre devuelven arrays de objetos Python, lo que puede ser un inconveniente. Por suerte, hay una función alternativa (pero menos rica en características), `numpy.vectorize`, que permite especificar el tipo de resultado:

```
In [142]: add_them = np.vectorize(add_elements, otypes=[np.float64])
```

```
In [143]: add_them(np.arange(8), np.arange(8))
Out[143]: array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14.])
```

Estas funciones ofrecen un modo de crear funciones similares a las universales, pero son muy lentas porque requieren una llamada a una función Python para calcular cada elemento, lo que resulta mucho más lento que los bucles ufunc basados en C de NumPy:

```
In [144]: arr = rng.standard_normal(10000)
```

```
In [145]: %timeit add_them(arr, arr)
2.43 ms +- 30.5 us per loop (mean +- std. dev. of 7 runs,
100 loops each)
```

```
In [146]: %timeit np.add(arr, arr)
```

```
2.88 us +- 47.9 ns per loop (mean +- std. dev. of 7 runs,
100000 loops each)
```

Un poco más adelante en este apéndice veremos cómo crear ufuncs rápidas en Python usando la librería Numba (<http://numba.pydata.org>).

A.5 Arrays estructurados y de registros

Por lo que hemos visto hasta ahora, podemos deducir que `ndarray` es un contenedor de datos homogéneo, es decir, representa un bloque de memoria en el que cada elemento ocupa el mismo número de bytes, tal y como determina el tipo de datos. Aparentemente puede parecer que esto nos impida representar datos heterogéneos o tabulares. Un array estructurado es un `ndarray` en el que cada elemento puede verse como un *struct* de C (de ahí

el nombre «estructurado») o como una fila en una tabla SQL con varios campos con nombre:

```
In [147]: dtype = [('x', np.float64), ('y', np.int32)]  
In [148]: sarr = np.array([(1.5, 6), (np.pi, -2)],  
dtype=dtype)  
In [149]: sarr  
Out[149]: array([(1.5, 6), (3.1416, -2)], dtype=[('x', '
```

Hay varias formas de especificar un tipo de datos estructurado (véase la documentación en línea de NumPy). Una habitual es como una lista de tuplas con (field_name, field_data_type). Ahora los elementos del array son objetos similares a una tupla a cuyos elementos puede accederse como a un diccionario:

```
In [150]: sarr[0]  
Out[150]: (1.5, 6)  
  
In [151]: sarr[0]['y']  
Out[151]: 6
```

Los nombres de los campos se almacenan en el atributo `dtype.names`. Cuando se accede a un campo del array estructurado, se devuelve una vista escalonada de los datos, lo que no copia nada:

```
In [152]: sarr['x']  
Out[152]: array([1.5, 3.1416])
```

Tipos de datos anidados y campos multidimensionales

Al especificar un tipo de datos estructurado, se puede pasar al mismo tiempo una forma (como un entero o una tupla):

```
In [153]: dtype = [('x', np.int64, 3), ('y', np.int32)]  
In [154]: arr = np.zeros(4, dtype=dtype)  
In [155]: arr
```

```
Out[155]:
```

```
array([[[0, 0, 0], 0), [[0, 0, 0], 0), [[0, 0, 0], 0), [[0, 0, 0], 0)],  
      dtype=[('x', '<i8', (3,)), ('y', '<i4')])
```

En este caso, el campo `x` hace ahora referencia a un array de longitud 3 para cada registro:

```
In [156]: arr[0]['x']  
Out[156]: array([0, 0, 0])
```

Resulta cómodo que acceder a `arr['x']` devuelva entonces un array bidimensional en lugar de uno de una sola dimensión, como en los ejemplos anteriores:

```
In [157]: arr['x']  
Out[157]:
```

```
array([ [0, 0, 0],  
        [0, 0, 0],  
        [0, 0, 0],  
        [0, 0, 0]])
```

Esto permite expresar estructuras anidadas más complicadas como un solo bloque de memoria en un array. También se pueden anidar los tipos de datos para crear estructuras más complejas. Aquí tenemos un ejemplo:

```
In [158]: dtype = [('x', [('a', 'f8'), ('b', 'f4')]), ('y', np.int32)]
```

```
In [159]: data = np.array([(1, 2), 5), ((3, 4), 6)],  
dtype=dtype)
```

```
In [160]: data['x']  
Out[160]: array([(1., 2.), (3., 4.)], dtype=[('a', '<f8'),  
('b', '<f4')])
```

```
In [161]: data['y']  
Out[161]: array([5, 6], dtype=int32)
```

```
In [162]: data['x']['a']
Out[162]: array([1., 3.])
```

El objeto DataFrame de pandas no soporta esta característica del mismo modo, aunque es similar a la indexación jerárquica.

¿Por qué emplear arrays estructurados?

Comparados con un objeto DataFrame de pandas, los arrays estructurados de NumPy son una herramienta de menor nivel. Ofrecen un medio para interpretar un bloque de memoria como una estructura tabular con columnas anidadas. Como cada elemento del array se representa en la memoria como un número de bytes fijo, los arrays estructurados ofrecen una manera eficaz de escribir datos en disco y recuperarlos de él (incluyendo los mapas de memoria), transportarlos por la red y otras aplicaciones similares. La disposición de memoria de cada valor de un array estructurado se basa en la representación binaria de los tipos de datos struct del lenguaje de programación C.

Como otro uso común de los arrays estructurados, escribir archivos de datos como flujos de bytes de registro de longitud fija es una forma habitual de serializar los datos en código C y C++, lo que en ocasiones se puede encontrar en el mercado en sistemas heredados. Siempre que el formato del archivo sea conocido (el tamaño de cada registro y el orden, el tamaño del byte y el tipo de datos de cada elemento), los datos se pueden leer en la memoria con `np.fromfile`. Aplicaciones especializadas como esta quedan fuera del alcance de este libro, pero vale la pena saber que estas cosas son posibles.

A.6 Más sobre la ordenación

Al igual que la lista interna de Python, el método de instancia `sort` del ndarray es una ordenación instantánea (*in-place*), lo que significa que el contenido del array es reordenado sin producir un nuevo array:

```
In [163]: arr = rng.standard_normal(6)
```

```
In [164]: arr.sort()
```

```
In [165]: arr
```

```
Out[165]: array([-1.1553, -0.9319, -0.5218, -0.4745,
 -0.1649, 0.03])
```

Cuando se ordenan arrays sin crear copias adicionales, conviene recordar que si el array es una vista de un ndarray distinto, el original resultará modificado:

```
In [166]: arr = rng.standard_normal((3, 5))
```

```
In [167]: arr
```

```
Out[167]:
```

```
array([[-1.1956, 0.4691, -0.3598, 1.0359, 0.2267],
 [-0.7448, -0.5931, -1.055, -0.0683, 0.458],
 [-0.07, 0.1462, -0.9944, 1.1436, 0.5026]])
```

```
In [168]: arr[:, 0].sort() # Ordena in-place los primeros
valores de columna
```

```
In [169]: arr
```

```
Out[169]:
```

```
array([[-1.1956, 0.4691, -0.3598, 1.0359, 0.2267],
 [-0.7448, -0.5931, -1.055, -0.0683, 0.458],
 [-0.07, 0.1462, -0.9944, 1.1436, 0.5026]])
```

Por otro lado, numpy.sort crea una copia nueva y ordenada de un array. De otro modo, acepta los mismos argumentos (como kind) que el método sort del ndarray:

```
In [170]: arr = rng.standard_normal(5)
```

```
In [171]: arr
```

```
Out[171]: array([ 0.8981, -1.1704, -0.2686, -0.796 ,
 1.4522])
```

```
In [172]: np.sort(arr)
```

```
Out[172]: array([-1.1704, -0.796, -0.2686, 0.8981, 1.4522])
```

```
In [173]: arr  
Out[173]: array([ 0.8981, -1.1704, -0.2686, -0.796 ,  
1.4522])
```

Todos estos métodos de ordenación toman un argumento de eje para ordenar de forma independiente las secciones de los datos a lo largo del eje pasado:

```
In [174]: arr = rng.standard_normal((3, 5))
```

```
In [175]: arr  
Out[175]:
```

```
array([ [-0.2535, 2.1183, 0.3634, -0.6245, 1.1279],  
       [ 1.6164, -0.2287, -0.6201, -0.1143, -1.2067],  
       [-1.0872, -2.1518, -0.6287, -1.3199, 0.083 ]])
```

```
In [176]: arr.sort(axis=1)
```

```
In [177]: arr  
Out[177]:
```

```
array([ [-0.6245, -0.2535, 0.3634, 1.1279, 2.1183],  
       [-1.2067, -0.6201, -0.2287, -0.1143, 1.6164],  
       [-2.1518, -1.3199, -1.0872, -0.6287, 0.083 ]])
```

Quizá alguno de mis lectores haya observado que ninguno de los métodos de ordenación tienen una opción para ordenar de forma descendente. Esto es un problema en la práctica, porque la segmentación de arrays produce vistas, y no crea, por lo tanto, copias ni requiere trabajo computacional alguno. Muchos usuarios de Python están familiarizados con el «truco» de que para una lista de values, values[::-1] devuelve una lista en orden inverso. Lo mismo aplica para los ndarrays:

```
In [178]: arr[:, ::-1]  
Out[178]:
```

```
array([ 2.1183, 1.1279, 0.3634, -0.2535, -0.6245],
```

```
[ 1.6164, -0.1143, -0.2287, -0.6201, -1.2067],  
[ 0.083 , -0.6287, -1.0872, -1.3199, -2.1518]])
```

Ordenaciones indirectas: argsort y lexsort

En análisis de datos es posible que surja la necesidad de reordenar conjuntos de datos según una o varias claves. Por ejemplo, si tenemos una tabla de datos de estudiantes, quizá necesitaríamos ordenarla primero por apellido y después por nombre. Es un ejemplo de una ordenación indirecta y, como ya vimos en los capítulos relacionados con pandas, existen muchos ejemplos de alto nivel. Dada una clave o claves (un array de valores o varios), deseamos obtener un array de índices enteros (me refiero a ellos coloquialmente como indexadores) que nos indique cómo recolocar los datos para que estén ordenados. Dos métodos para llevar esto a cabo son argsort y numpy.lexsort. Como ejemplo tenemos lo siguiente:

```
In [179]: values = np.array([5, 0, 1, 3, 2])
```

```
In [180]: indexer = values.argsort()
```

```
In [181]: indexer
```

```
Out[181]: array([1, 2, 4, 3, 0])
```

```
In [182]: values[indexer]
```

```
Out[182]: array([0, 1, 2, 3, 5])
```

Este código que muestro a continuación, como ejemplo más complicado, reordena un array bidimensional según su primera fila:

```
In [183]: arr = rng.standard_normal((3, 5))
```

```
In [184]: arr[0] = values
```

```
In [185]: arr
```

```
Out[185]:
```

```
array([[ 5. ,  0. ,  1. ,  3. ,  2. ],  
       [-0.7503, -2.1268, -1.391 , -0.4922,  0.4505],  
       [ 0.8926, -1.0479,  0.9553,  0.2936,  0.5379]])
```

```
In [186]: arr[:, arr[0].argsort()]
Out[186]:
```

```
array([  0. ,  1. ,  2. ,  3. ,  5. ],
      [-2.1268, -1.391 ,  0.4505, -0.4922, -0.7503],
      [-1.0479,  0.9553,  0.5379,  0.2936,  0.8926]])
```

La función `lexsort` es similar a `argsort`, pero realiza una ordenación lexicográfica indirecta según arrays de varias claves. Supongamos que queremos ordenar algunos datos identificados por nombres y apellidos:

```
In [187]: first_name = np.array(['Bob', 'Jane', 'Steve',
                                'Bill', 'Barbara'])
```

```
In [188]: last_name = np.array(['Jones', 'Arnold', 'Arnold',
                               'Jones', 'Walters'])
```

```
In [189]: sorter = np.lexsort((first_name, last_name))
```

```
In [190]: sorter
Out[190]: array([1, 2, 3, 0, 4])
```

```
In [191]: list(zip(last_name[sorter], first_name[sorter]))
Out[191]:
```

```
[('Arnold', 'Jane'),
 ('Arnold', 'Steve'),
 ('Jones', 'Bill'),
 ('Jones', 'Bob'),
 ('Walters', 'Barbara')]
```

La primera vez que se utiliza `lexsort` resulta un poco confuso, porque el orden en el que se emplean las claves para ordenar los datos empieza por el último array pasado. Aquí, `last_name` se usó antes que `first_name`.

Algoritmos de ordenación alternativos

Un algoritmo de ordenación estable conserva la posición relativa de elementos iguales. Esto puede ser de especial importancia en ordenaciones

indirectas, en las que el orden relativo es significativo:

```
In [192]: values = np.array(['2:first',    '2:second',
 '1:first', '1:second',
 ....: '1:third'])

In [193]: key = np.array([2, 2, 1, 1, 1])

In [194]: indexer = key.argsort(kind='mergesort')

In [195]: indexer
Out[195]: array([2, 3, 4, 0, 1])

In [196]: values.take(indexer)

Out[196]:
array(['1:first',     '1:second',     '1:third',     '2:first',
 '2:second'],
      dtype='<U8')
```

El único método de orden estable es el ordenamiento por mezcla («*mergesort*»), que tiene garantizado un rendimiento $O(n \log n)$, pero en promedio su rendimiento es peor que el del método predeterminado *quicksort*.

En la tabla A.3 podemos consultar un resumen de los métodos disponibles y sus rendimientos relativos (y garantías de rendimiento). Esto no es algo en lo que la mayoría de los usuarios tenga que pensar, pero resulta útil saber que existe.

Tabla A.3. Métodos de ordenación de arrays.

Tipo	Velocidad	Estable	Espacio de trabajo	Caso peor
'quicksort'	1	No	0	$O(n^2)$
'mergesort'	2	Sí	$n / 2$	$O(n \log n)$
'heapsort'	3	No	0	$O(n \log n)$

Ordenación parcial de arrays

Uno de los objetivos de la ordenación puede ser determinar los elementos mayor y menor de un array. NumPy tiene dos métodos rápidos, `numpy.partition` y `np.argpartition`, para realizar particiones en un array en torno al k elemento menor:

```
In [197]: rng = np.random.default_rng(12345)
```

```
In [198]: arr = rng.standard_normal(20)
```

```
In [199]: arr
```

```
Out[199]:
```

```
array([-1.4238,  1.2637, -0.8707, -0.2592, -0.0753,
       -0.7409, -1.3678,
       0.6489,  0.3611, -1.9529,  2.3474,  0.9685, -0.7594,
       0.9022,
      -0.467 , -0.0607,  0.7888, -1.2567,  0.5759,  1.399
     ])
```

```
In [200]: np.partition(arr, 3)
```

```
Out[200]:
```

```
array([-1.9529, -1.4238, -1.3678, -1.2567, -0.8707,
       -0.7594, -0.7409,
      -0.0607,  0.3611, -0.0753, -0.2592, -0.467 ,
       0.5759,  0.9022,
       0.9685,  0.6489,  0.7888,  1.2637,  1.399 ,  2.3474])
```

Después de llamar a `partition(arr, 3)`, los primeros tres elementos del resultado son los tres valores menores sin ningún orden especial. La función `numpy.argpartition`, similar a `numpy.argsort`, devuelve los índices que reordenan los datos en el orden equivalente:

```
In [201]: indices = np.argpartition(arr, 3)
```

```
In [202]: indices
```

```
Out[202]:
```

```
array([ 9,  0,  6, 17,  2, 12,  5, 15,  8,  4,  3, 14, 18, 13,
```

```
    11,  7,  16,
    1,  19, 10])
```

```
In [203]: arr.take(indices)
Out[203]:
```

```
array([-1.9529, -1.4238, -1.3678, -1.2567, -0.8707,
       -0.7594, -0.7409,
       -0.0607,  0.3611, -0.0753, -0.2592, -0.467 ,
       0.5759,  0.9022,
       0.9685,  0.6489,  0.7888,  1.2637,  1.399 ,  2.3474])
```

Localización de elementos en un array ordenado con numpy.searchsorted

El método array searchsorted realiza una búsqueda binaria en un array ordenado, devolviendo la posición del array en la que el valor tendría que ser insertado para mantener la ordenación:

```
In [204]: arr = np.array([0, 1, 7, 12, 15])
In [205]: arr.searchsorted(9)
Out[205]: 3
```

También se puede pasar un array de valores para obtener de vuelta un array de índices:

```
In [206]: arr.searchsorted([0, 8, 11, 16])
Out[206]: array([0, 3, 3, 5])
```

La función searchsorted devolvió 0 para el elemento 0, porque el comportamiento predeterminado es devolver el índice de la parte izquierda de un grupo de valores iguales:

```
In [207]: arr = np.array([0, 0, 0, 1, 1, 1, 1])
In [208]: arr.searchsorted([0, 1])
Out[208]: array([0, 3])
In [209]: arr.searchsorted([0, 1], side='right')
Out[209]: array([3, 7])
```

Como otra aplicación de `searchsorted`, supongamos que tenemos un array de valores entre 0 y 10.000, y un array aparte de «bordes de contenedor» que queremos usar para guardar los datos:

```
In [210]: data = np.floor(rng.uniform(0, 10000, size=50))
```

```
In [211]: bins = np.array([0, 100, 1000, 5000, 10000])
```

```
In [212]: data
```

```
Out[212]:
```

```
array([ 815., 1598., 3401., 4651., 2664., 8157., 1932.,
       1294., 916.,
       5985., 8547., 6016., 9319., 7247., 8605., 9293.,
       5461., 9376.,
       4949., 2737., 4517., 6650., 3308., 9034., 2570.,
       3398., 2588.,
       3554., 50., 6286., 2823., 680., 6168., 1763.,
       3043., 4408.,
       1502., 2179., 4743., 4763., 2552., 2975., 2790.,
       2605., 4827.,
       2119., 4956., 2462., 8384., 1801.])
```

Para obtener después un etiquetado de a qué intervalo pertenece cada punto de datos (donde 1 significaría el contenedor [0, 100)), podemos sencillamente usar `searchsorted`:

```
In [213]: labels = bins.searchsorted(data)
```

```
In [214]: labels
```

```
Out[214]:
```

```
array([2, 3, 3, 3, 3, 4, 3, 3, 2, 4, 4, 4, 4, 4, 4, 4,
       4, 4, 3, 3, 3, 4,
       3, 4, 3, 3, 3, 3, 1, 4, 3, 2, 4, 3, 3, 3, 3, 3,
       3, 3, 3, 3, 3,
       3, 3, 3, 4, 3])
```

Esto, combinado con `groupby` de pandas, se puede emplear para guardar los datos:

```
In [215]: pd.Series(data).groupby(labels).mean()
Out[215]:
```

		50.000000
1		
2		803.666667
3		3079.741935
4		7635.200000

dtype: float64

A.7 Escritura de funciones rápidas NumPy con Numba

Numba (<http://numba.pydata.org>) es un proyecto de código abierto que crea funciones rápidas para datos similares a los de NumPy utilizando CPU, GPU u otro hardware. Emplea el proyecto LLVM (<http://llvm.org/>) para traducir código Python en código máquina compilado.

Para presentar Numba, veamos una función pura de Python que calcula la expresión $(x-y)$. `mean()` usando un bucle `for`:

```
import numpy as np
def mean_distance(x, y):

    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i]-y[i]
        count += 1
    return result / count
```

Esta función es lenta:

```
In [209]: x = rng.standard_normal(10_000_000)
In [210]: y = rng.standard_normal(10_000_000)
In [211]: %timeit mean_distance(x, y)
```

```
1 loop, best of 3: 2 s per loop
In [212]: %timeit (x-y).mean()
100 loops, best of 3: 14.7 ms per loop
```

La versión de NumPy es más de 100 veces más rápida. Podemos convertir esta función en una compilada de Numba utilizando la función `numba.jit`:

```
In [213]: import numba as nb
In [214]: numba_mean_distance = nb.jit(mean_distance)
```

También podríamos haber escrito esto como un decorador:

```
@nb.jit

def numba_mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i]-y[i]
        count += 1
    return result / count
```

La función resultante es de hecho más rápida que la versión vectorizada de NumPy:

```
In [215]: %timeit numba_mean_distance(x, y)
100 loops, best of 3: 10.3 ms per loop
```

Numba no es capaz de compilar todo el código puro de Python, pero soporta un importante subconjunto del lenguaje, lo cual resulta de máxima utilidad para escribir algoritmos numéricos.

Numba es una librería profunda, que soporta distintos tipos de hardware, modos de compilación y extensiones de usuario. También puede compilar un considerable subconjunto de la API Python de NumPy sin bucles `for` explícitos. Numba es capaz de reconocer construcciones que se pueden

compilar a código máquina, y sustituye llamadas a la API de CPython por funciones que no sabe cómo compilar. La opción de la función `jit` de Numba, `nopython=True`, restringe el código permitido a código de Python que puede compilarse a LLVM sin llamadas a la API C de Python. La opción `jit(nopython=True)` tiene una denominación reducida, `numba.njit`.

En el ejemplo anterior podríamos haber escrito:

```
from numba import float64, njit
@njit(float64(float64[:, :], float64[:]))
def mean_distance(x, y):

    return (x-y).mean()
```

Animo a mis lectores a obtener más información leyendo la documentación en línea de Numba (<http://numba.pydata.org/>). La siguiente sección muestra un ejemplo de la creación de objetos ufunc personalizados de NumPy.

Creación de objetos personalizados numpy.ufunc con Numba

La función `numba.vectorize` crea ufuncs compiladas de NumPy, que se comportan como las ufuncs internas. Veamos una implementación Python de `numpy.add`:

```
from numba import vectorize
@vectorize

def nb_add(x, y):
    return x + y
```

Ahora tenemos:

```
In [13]: x = np.arange(10)
```

```
In [14]: nb_add(x, x)
```

```
Out[14]: array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14., 16.,
18.])

In [15]: nb_add.accumulate(x, 0)
Out[15]: array([ 0.,  1.,  3.,  6., 10., 15., 21., 28., 36.,
45.])
```

A.8 Entrada y salida de arrays avanzadas

En el capítulo 4 nos habituamos al uso de `np.save` y `np.load` para almacenar arrays en formato binario en disco. Hay otras opciones adicionales que se pueden considerar para un uso más sofisticado. En particular, los mapas de memoria tienen el beneficio adicional de permitirnos realizar determinadas operaciones con conjuntos de datos que no caben en la RAM.

Archivos mapeados en memoria

Un archivo proyectado o mapeado en memoria es un método para interactuar con datos binarios en disco como si estuvieran almacenados en un array dentro de la memoria. NumPy implementa un objeto `memmap` que es del tipo `ndarray`, y que permite leer y grabar pequeños segmentos de un archivo grande sin tener que leer el array completo en la memoria. Además, un `memmap` tiene los mismos métodos que un array en memoria y por tanto se puede sustituir en muchos algoritmos en los que se esperaría un `ndarray`.

Para crear un nuevo mapa de memoria, usamos la función `np.memmap` y pasamos una ruta de archivo, un tipo de datos, una forma y un modo de archivo:

```
In [217]: mmap = np.memmap('mymmap', dtype='float64',
mode='w+',
.....: shape=(10000, 10000))

In [218]: mmap
Out[218]:
```

```
memmap([          [0., 0., 0., ..., 0., 0., 0.],  
              [0., 0., 0., ..., 0., 0., 0.],  
              [0., 0., 0., ..., 0., 0., 0.],  
              ...,  
              [0., 0., 0., ..., 0., 0., 0.],  
              [0., 0., 0., ..., 0., 0., 0.],  
              [0., 0., 0., ..., 0., 0., 0.]])
```

Segmentar un `memmap` devuelve vistas de los datos en disco:

```
In [219]: section = mmap[:5]
```

Si se les asignan datos, las vistas serán almacenadas en la memoria, lo que significa que los cambios no se verán reflejados inmediatamente en el archivo en disco si lo leyéramos en una aplicación distinta. Las modificaciones se pueden sincronizar en disco llamando a `flush`:

```
In [220]: section[:] = rng.standard_normal((5, 10000))
```

```
In [221]: mmap.flush()
```

```
In [222]: mmap
```

```
Out[222]:
```

```
memmap([      [-0.9074, -1.0954,  0.0071, ...,  0.2753,  
                 -1.1641,  0.8521],  
                 [-0.0103, -0.0646, -1.0615, ..., -1.1003,  
                  0.2505,  0.5832],  
                 [ 0.4583,  1.2992,  1.7137, ...,  0.8691, -0.7889,  
                  -0.2431],  
                 ...,  
                 [ 0. ,  0. ,  0. , ..., 0. ,  0. ,  0. ],  
                 [ 0. ,  0. ,  0. , ..., 0. ,  0. ,  0. ],  
                 [ 0. ,  0. ,  0. , ..., 0. ,  0. ,  0. ]])
```

```
In [223]: del mmap
```

Siempre que un mapa de memoria quede fuera de alcance y sea eliminado, los cambios que se hayan producido también serán descartados.

Al abrir un mapa de memoria existente, sigue siendo necesario especificar el tipo y la forma de los datos, porque el archivo es solamente un bloque de datos binarios sin información alguna sobre el tipo de datos, la forma o los escalones:

```
In [224]: mmap = np.memmap('mymmap', dtype='float64', shape=(10000, 10000))
```

```
In [225]: mmap
```

```
Out[225]:
```

```
memmap([ [-0.9074, -1.0954, 0.0071, ..., 0.2753,
           -1.1641, 0.8521],
          [-0.0103, -0.0646, -1.0615, ..., -1.1003,
           0.2505, 0.5832],
          [ 0.4583, 1.2992, 1.7137, ..., 0.8691, -0.7889,
           -0.2431],
          ...,
          [ 0., 0., 0., ..., 0., 0., 0. ],
          [ 0., 0., 0., ..., 0., 0., 0. ],
          [ 0., 0., 0., ..., 0., 0., 0. ]])
```

Los mapas de memoria también funcionan con tipos de datos estructurados o anidados, como se describe en la sección A.5 «Arrays estructurados y de registros».

Si ejecutáramos este ejemplo en nuestro ordenador, quizás nos interesaría borrar el enorme archivo que habíamos creado antes:

```
In [226]: %xdel mmap
```

```
In [227]: !rm my mmap
```

HDF5 y otras opciones de almacenamiento de arrays

PyTables y h5py son dos proyectos de Python que ofrecen interfaces válidos con NumPy para almacenar datos de arrays en el formato HDF5, eficaz y comprimible (HDF significa *Hierarchical Data Format*: formato de datos jerárquico). Se pueden almacenar con total seguridad cientos de

gigabytes o incluso terabytes de datos en formato HDF5. Para saber más sobre el uso de este formato con Python, recomiendo la lectura de la documentación en línea de pandas (<http://pandas.pydata.org>).

A.9 Consejos de rendimiento

La adaptación del código de procesamiento de datos para su uso con NumPy suele acelerar bastante las cosas, ya que las operaciones con arrays reemplazan normalmente los bucles puros de Python, de otro modo extremadamente lentos. Aquí ofrezco algunos consejos para obtener el mejor rendimiento de la librería:

- Convertir los bucles de Python y la lógica condicional en operaciones con arrays y arrays booleanos.
- Utilizar la difusión siempre que sea posible.
- Emplear vistas de arrays (segmentación) para evitar la copia de datos.
- Utilizar funciones universales y métodos ufunc.

Si no se puede obtener el rendimiento deseado tras agotar las posibilidades que ofrece

NumPy por sí solo, una opción es escribir código C, FORTRAN o Cython. Yo utilizo Cython (<http://cython.org>) con frecuencia en mi trabajo como un modo de obtener un rendimiento similar al de C, a menudo con un tiempo de desarrollo mucho menor.

La importancia de la memoria contigua

Aunque tratar este tema a fondo quede fuera del alcance de este libro, en algunas aplicaciones la disposición en la memoria de un array puede afectar de forma significativa a la velocidad de los cálculos. Esto se basa en parte en la diferencia de rendimiento asociada a la jerarquía de caché de la CPU; las operaciones que acceden a bloques de memoria contiguos (por ejemplo, sumar las filas de un array en orden de C) serán en general las más rápidas, porque el subsistema de la memoria almacenará los bloques de memoria

adecuados en las cachés L1 o L2 de baja latencia de la CPU. Además, hay determinadas rutas de código dentro del código base C de NumPy que han sido optimizadas para el caso contiguo, en el que se puede evitar el acceso genérico a la memoria escalonada.

Decir que la disposición en la memoria de un array es contigua significa que los elementos se almacenan en la memoria en el orden en el que aparecen en el array con respecto al orden de FORTRAN (principal de columna) o de C (principal de fila). De forma predeterminada, los arrays NumPy se crean como contiguos de C o simplemente contiguos. Un array principal de columna, como la transposición de un array contiguo de C, se dice, por tanto, que es contiguo de FORTRAN. Estas propiedades pueden verificarse de forma explícita mediante el atributo `flags` del ndarray:

```
In [228]: arr_c = np.ones((100, 10000), order='C')
```

```
In [229]: arr_f = np.ones((100, 10000), order='F')
```

```
In [230]: arr_c.flags  
Out[230]:
```

```
C_CONTIGUOUS : True  
F_CONTIGUOUS : False  
OWNDATA : True  
WRITEABLE : True  
ALIGNED : True  
WRITEBACKIFCOPY : False  
UPDATEIFCOPY : False
```

```
In [231]: arr_f.flags  
Out[231]:
```

```
C_CONTIGUOUS : False  
F_CONTIGUOUS : True  
OWNDATA : True  
WRITEABLE : True  
ALIGNED : True  
WRITEBACKIFCOPY : False  
UPDATEIFCOPY : False
```

```
In [232]: arr_f.flags.f_contiguous
Out[232]: True
```

En este ejemplo, sumar las filas de estos arrays debería ser, en teoría, más rápido para arr_c que para arr_f, ya que las filas son contiguas en la memoria. Aquí compruebo usando %timeit en IPython (estos resultados pueden ser distintos en otras máquinas):

```
In [233]: %timeit arr_c.sum(1)
444 us +- 60.5 us per loop (mean +- std. dev. of 7 runs,
1000 loops each)
```

```
In [234]: %timeit arr_f.sum(1)
```

```
581 us +- 8.16 us per loop (mean +- std. dev. of 7 runs,
1000 loops each)
```

Cuando lo que buscamos es sacar más rendimiento de NumPy, con frecuencia suele ser necesario en este caso invertir un cierto esfuerzo. Si el array de que disponemos no tiene el orden de memoria deseado, se puede utilizar copy y pasar 'C' o 'F':

```
In [235]: arr_f.copy('C').flags
Out[235]:
```

```
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
UPDATEIFCOPY : False
```

Al construir una vista en un array, conviene recordar que no se garantiza que el resultado sea contiguo:

```
In [236]: arr_c[:50].flags.contiguous
Out[236]: True
```

```
In [237]: arr_c[:, :50].flags
```

Out[237]:

```
C_CONTIGUOUS : False
F_CONTIGUOUS : False
OWNDATA : False
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
UPDATEIFCOPY : False
```

⁴ Algunos de los tipos de datos tienen un signo de subrayado al final de sus nombres. Este signo está ahí para evitar conflictos de nombre de variable entre los tipos específicos de NumPy y los internos de Python.

Más sobre el sistema IPython

En el capítulo 2 vimos los fundamentos del manejo del shell de IPython y del notebook de Jupyter. En este apéndice vamos a explorar otras funcionalidades avanzadas del sistema IPython que se pueden utilizar desde la consola o desde dentro de Jupyter.

B.1 Atajos de teclado del terminal

IPython tiene muchos atajos de teclado (que les resultarán familiares a los usuarios del editor de texto Emacs o del shell Bash de Unix) para desplazarse en el prompt e interactuar con el historial de comandos del shell. La tabla B.1 resume algunos de los atajos más usados. En la figura B.1 se visualizan también algunos, como, por ejemplo, el movimiento del cursor.

Tabla B.1. Atajos de teclado IPython estándares.

Atajo de teclado	Descripción
Control-P o flecha arriba	Busca hacia atrás en el historial comandos que empiecen por el texto introducido.
Control-N o flecha abajo	Busca hacia adelante en el historial comandos que empiecen por el texto introducido.
Control-R	Búsqueda inversa en el historial al estilo Readline (coincidencia parcial).
Control-Mayús-V	Pega texto del portapapeles.
Control-C	Interrumpe código en ejecución.
Control-A	Mueve el cursor al comienzo de la línea.
Control-E	Mueve el cursor al final de la línea.
Control-K	Borra texto del cursor hasta el final de la línea.
Control-U	Descarta todo el texto de la línea actual.
Control-F	Mueve el cursor hacia delante un carácter.
Control-B	Mueve el cursor hacia atrás un carácter.
Control-L	Vacia la pantalla.



Figura B.1. Ilustración de algunos atajos de teclado del shell de IPython.

Los notebooks de Jupyter tienen un conjunto de atajos de teclado, en gran medida independiente, para navegación y edición. Como estos atajos han evolucionado a mayor velocidad que los de IPython, animo a mis lectores a que exploren el sistema de ayuda integrado de los menús del notebook de Jupyter.

B.2 Los comandos mágicos

Los comandos especiales de IPython (que no están incluidos en el propio Python) se denominan comandos mágicos. Están diseñados para facilitar tareas habituales y permitir controlar fácilmente el comportamiento del sistema IPython. Un comando mágico es cualquier comando precedido por el símbolo del porcentaje (%). Por ejemplo, se puede comprobar la ejecución de cualquier sentencia Python, como, por ejemplo, una multiplicación de matrices, usando la función mágica %timeit:

```
In [20]: a = np.random.standard_normal((100, 100))
```

```
In [20]: %timeit np.dot(a, a)
92.5 µs ± 3.43 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Los comandos mágicos se pueden visualizar como programas de línea de comando que se ejecutan dentro del sistema IPython. Muchos de ellos tienen opciones adicionales de «línea de comando», las cuales se pueden ver (como era de esperar) utilizando ?:

```
In [21]: %debug?
```

Docstring:

```
::
%debug [-breakpoint FILE:LINE] [statement [statement ...]]
Activate the interactive debugger.
This magic command support two ways of activating debugger.
One is to activate debugger before executing code. This way, you
can set a break point, to step through the code from the point.
You can use this mode by giving statements to execute and
optionally a breakpoint.
The other one is to activate debugger in post-mortem mode. You can
activate this mode simply running %debug without any argument.
```

```
If an exception has just occurred, this lets you inspect its stack
frames interactively. Note that this will always work only on the
last traceback that occurred, so you must call this quickly after
an exception that you wish to inspect has fired, because if
another one occurs, it clobbers the previous one.
If you want IPython to automatically do this on every exception,
see the %pdb magic for more details.
```

```
.. versionchanged:: 7.3
When running code, user variables are no longer expanded,
the magic line is always left unmodified.
positional arguments:
statement Code to run in debugger. You can omit this
in cell magic mode.
```

```

optional arguments:
  -breakpoint <FILE:LINE>, -b <FILE:LINE>
Set break point at LINE in FILE.

```

Las funciones mágicas se pueden utilizar de forma predeterminada sin el signo del porcentaje, siempre que no haya ninguna otra variable definida con el mismo nombre aparte de la función mágica en cuestión. A esta característica se le denomina *automagic* y se puede habilitar o deshabilitar con `%automagic`.

Algunas funciones mágicas se comportan como funciones Python, y su resultado puede asignarse a una variable:

```

In [22]: %pwd
Out[22]: '/home/wesm/code/pydata-book'

In [23]: foo = %pwd

In [24]: foo
Out[24]: '/home/wesm/code/pydata-book'

```

Como es posible acceder a la documentación de IPython desde dentro del sistema, es muy interesante explorar todos los comandos especiales disponibles utilizando `%quickref` o `%magic`. Esta información se muestra en una ventana aparte, de modo que será necesario pulsar `q` para salir de ella. La tabla B.2 lista algunos de los comandos básicos que es necesario conocer para ser productivos en computación interactiva y en desarrollo de Python trabajando en IPython.

Tabla B.2. Algunos de los comandos mágicos IPython más utilizados.

Comando	Descripción
<code>%quickref</code>	Muestra la guía de referencia rápida de IPython.
<code>%magic</code>	Muestra documentación detallada de todos los comandos mágicos disponibles.
<code>%debug</code>	Entra en el depurador interactivo al final del último rastreo de excepciones.
<code>%hist</code>	Imprime el historial de comandos introducidos en la línea de comandos (y de forma opcional los resultados).
<code>%pdb</code>	Entra automáticamente en el depurador después de cualquier excepción.
<code>%paste</code>	Ejecuta código Python preformatado desde el portapapeles.
<code>%cpaste</code>	Abre un prompt especial para pegar manualmente código Python que se va a ejecutar.
<code>%reset</code>	Borra todas las variables o nombres definidos en un espacio de nombres interactivo.
<code>%page</code> <i>OBJETO</i>	Imprime el objeto y lo muestra a través de una ventana aparte.
<code>%run</code> <i>script.py</i>	Ejecuta un script de Python dentro de IPython.
<code>%prun</code> <i>sentencia</i>	Ejecuta <i>sentencia</i> con <code>cProfile</code> e informa del resultado del perfilador.
<code>%time</code> <i>sentencia</i>	Informa del tiempo de ejecución de una sola sentencia.
<code>%timeit</code> <i>sentencia</i>	Ejecuta una sentencia varias veces para calcular un tiempo de ejecución medio en conjunto; es útil para cronometrar el tiempo de ejecución de código cuando es muy corto.

Comando	Descripción
%who, %who_ls, %whos	Muestra variables definidas en el espacio de nombres interactivo, con niveles variables de informacion o verbosidad.
%xdel <i>variable</i>	Borra una variable e intenta eliminar cualquier referencia al objeto en el interior de IPython.

El comando %run

Se puede ejecutar cualquier archivo como un programa Python dentro del entorno de la sesión de IPython en la que nos encontremos mediante el comando `%run`. Supongamos que tenemos el siguiente código almacenado en `script.py`:

```
def f(x, y, z):
    return (x + y) / z

a = 5
b = 6
c = 7.5
result = f(a, b, c)
```

Podemos ejecutar esto pasándole a `%run` el nombre del archivo:

```
In [14]: %run script.py
```

El script se ejecuta en un espacio de nombres vacío (sin importaciones u otras variables definidas), de modo que el comportamiento debe ser idéntico a la ejecución del programa en la línea de comandos utilizando `python script.py`. Todas las variables (importaciones, funciones y globales) definidas en el archivo (hasta que se produzca una excepción, si es que la hay) estarán entonces accesibles en el shell de IPython:

```
In [15]: c
Out [15]: 7.5
```

```
In [16]: result
Out[16]: 1.4666666666666666
```

Si un script de Python espera argumentos de línea de comando (que se pueden encontrar en `sys.argv`), se le pueden pasar después de la ruta del archivo como si se ejecutara en la línea de comandos.



Si se desea dar un acceso de script a variables ya definidas en el espacio de nombres IPython interactivo, es mejor usar `%run -i` en lugar del `%run` sencillo.

En un notebook de Jupyter, también podemos emplear la función mágica asociada `%load`, que importa un script en una celda de código:

```
In [16]: %load script.py
```

```
def f(x, y, z):
    return (x + y) / z
a = 5
b = 6

c = 7.5
result = f(a, b, c)
```

Interrumpir código en ejecución

Pulsar Control-C mientras cualquier código se está ejecutando, ya sea un script mediante `%run` o un comando de ejecución larga, producirá un error `KeyboardInterrupt`, lo cual provocará que casi todos los programas Python se detengan inmediatamente, excepto en ciertos casos poco habituales.



Cuando se ha llamado a un fragmento de código Python en algún módulo de extensión compilado, pulsar Control-C no siempre provoca que la ejecución del programa se detenga inmediatamente. En estos casos, habrá que esperar hasta que el intérprete de Python recupere el control o, en circunstancias más extremas, terminar forzadamente el proceso Python en el sistema operativo (como, por ejemplo, utilizando el Administrador de Tareas en Windows o el comando `kill` en Linux).

Ejecutar código desde el portapapeles

Si estamos utilizando el notebook de Jupyter, podemos copiar y pegar código en cualquier celda de código y ejecutarlo. Es también posible ejecutar código desde el portapapeles en el shell de IPython. Supongamos que tenemos el siguiente código en otra aplicación cualquiera:

```
x = 5
y = 7
if x > 5:

    x += 1
    y = 8
```

Los métodos más infalibles son las funciones mágicas `%paste` y `%cpaste` (tengamos en cuenta que no funcionan en Jupyter, ya que es posible copiar y pegar en una celda de código Jupyter).

La función `%paste` toma el texto que haya en el portapapeles y lo ejecuta como un solo bloque en el shell:

```
In [17]: %paste
x = 5
y = 7
if x > 5:

    x += 1
    y = 8

## - Fin del texto pegado -
```

La otra función `%cpaste` es similar, salvo que ofrece un prompt especial en el que pegar el código:

```
In [18]: %cpaste
Pasting code; enter '-' alone on the line to stop or use Ctrl-D.
:x = 5
```

```
:y = 7
:if x > 5:
: x += 1
:
: y = 8
:-
```

Con el bloque %cpaste, tenemos la libertad de pegar tanto código como queramos antes de ejecutarlo. Podríamos decidirnos por usar %cpaste para ver el código pegado antes de ejecutarlo. Si pegamos accidentalmente el código equivocado, podemos salir del prompt %cpaste pulsando Control-C.

B.3 Cómo utilizar el historial de comandos

IPython mantiene una pequeña base de datos en disco que contiene el texto de cada comando que se ejecuta. Esto sirve para varios propósitos:

- Buscar, completar y ejecutar comandos anteriormente ejecutados escribiendo lo mínimo en el teclado.
- Conservar el historial de comandos entre sesiones.
- Guardar el historial de entrada y salida en un archivo.

Estas funciones son más útiles en el shell que en el notebook, ya que este último, tal y como está diseñado, mantiene un registro de la entrada y la salida en cada celda de código.

Búsqueda y reutilización del historial de comandos

El shell de IPython permite buscar y ejecutar código anterior u otros comandos. Esto resulta útil, pues en ocasiones podemos descubrirnos repitiendo los mismos comandos, como un comando %run u otro fragmento de código cualquiera. Supongamos que hemos ejecutado lo siguiente:

```
In[7]: %run first/second/third/data_script.py
```

Después hemos explorado los resultados del script (suponiendo que se ejecutara sin errores) para descubrir que hicimos un cálculo incorrecto. Tras encontrar el problema y modificar data_script.py, se puede empezar tecleando algunas letras del comando %run y pulsando después la combinación de teclas Control-P o la tecla de la flecha arriba. Así, buscamos en el historial de comandos el primer comando previo que coincida con las letras escritas. Si se pulsa Control-P o la tecla arriba varias veces se sigue buscando a lo largo de la historia. No hay problema si saltamos el comando que queremos ejecutar, porque podemos avanzar en el historial pulsando Control-N o la flecha abajo. Tras hacer esto varias veces, probablemente lleguemos a pulsar estas teclas sin pensar.

Utilizar Control-R ofrece la misma capacidad de búsqueda incremental parcial ofrecida por el readline empleado en shells de estilo Unix, como Bash. En Windows, IPython se encarga de emular la funcionalidad readline. Para utilizarla, pulsamos Control-R y después escribimos varios caracteres contenidos en la línea de entrada en la que se deseé buscar:

```
In [1]: a_command = foo(x, y, z)
(reverse-i-search)`com': a_command = foo(x, y, z)
```

Al pulsar Control-R se recorre el historial línea a línea, comparando los caracteres escritos.

Variables de entrada y salida

Olvidarse de asignar a una variable el resultado de una llamada a una función puede ser un incordio. Una sesión de IPython almacena las referencias, tanto a los comandos de entrada como a los objetos Python de salida en variables especiales. Las dos salidas anteriores se almacenan en las variables `_` (un guion bajo) y `__` (dos guiones bajos), respectivamente:

```
In [18]: 'input1'  
Out[18]: 'input1'  
  
In [19]: 'input2'  
Out[19]: 'input2'  
  
In [20]: __  
Out[20]: 'input1'  
  
In [21]: 'input3'  
Out[21]: 'input3'  
  
In [22]: __  
Out[22]: 'input3'
```

Las variables de entrada se almacenan en variables llamadas `_ix`, donde `x` es el número de línea de entrada.

Para cada variable de entrada hay una correspondiente `_x` de salida. Así, después de la línea de entrada 27, por ejemplo, habrá dos variables nuevas, `_27` para la salida e `_i27` para la entrada:

```
In [26]: foo = 'bar'  
  
In [27]: foo  
Out[27]: 'bar'  
  
In [28]: _i27  
Out[28]: u'foo'  
  
In [29]: _27  
Out[29]: 'bar'
```

Como las variables de entrada son cadenas de texto, se pueden ejecutar de nuevo con la palabra clave `eval` de Python:

```
In [30]: eval(_i27)  
Out[30]: 'bar'
```

Aquí, `_i27` se refiere a la entrada de código de `In [27]`.

Existen varias funciones mágicas que permiten trabajar con el historial de entrada y salida. La función `%hist` imprime todo el historial de entrada o parte de él, con o sin números de línea, mientras que `%reset` borra el espacio de nombres interactivo y las cachés de entrada y salida de forma opcional. La función mágica `%xdel` elimina todas las referencias a un objeto en particular de la maquinaria IPython. En la documentación se puede encontrar más información sobre estas funciones.



Cuando se trabaja con conjuntos de datos muy grandes, conviene recordar que el historial de entrada y salida de IPython puede provocar que los objetos a los que en él se hace referencia no sean descartados (liberando la memoria), incluso aunque se borran las variables del espacio de nombres interactivo utilizando la palabra clave `del`. En estos casos, el uso prudente de `%xdel` y `%reset` puede ayudar a evitar problemas con la memoria.

B.4 Interacción con el sistema operativo

Otra característica de IPython es que permite acceder al sistema de archivos y al shell del sistema operativo. Esto significa, entre otras cosas, que se pueden realizar la mayoría de las acciones de línea de comando estándares como se haría en el shell de Windows o Unix (Linux, macOS) sin tener que salir de IPython. Se incluyen los comandos del shell, el cambio de directorios y el almacenamiento de los resultados de un comando en un objeto Python (lista o cadena de texto). Existen, asimismo, funciones de creación de alias de comandos y marcadores a directorios.

En la tabla B.3 podemos ver un resumen de las funciones mágicas y su sintaxis para llamar a comandos del shell. En las siguientes secciones examinaré brevemente todo esto.

Tabla B.3. Comandos de IPython relacionados con el sistema.

Comando	Descripción
!cmd	Ejecuta cmd en el shell del sistema.
output = !cmd args	Ejecuta cmd y almacena el stdout en output.
%alias nombre_de_alias cmd	Define un alias para un comando de sistema (shell).
%bookmark	Utiliza el sistema de marcadores a directorios de IPython.
%cd directorio	Cambia el directorio de trabajo del sistema al directorio que se haya pasado.
%pwd	Vuelve al directorio de trabajo actual del sistema.
%pushd directorio	Pone el directorio actual en la pila y cambia al directorio de destino.
%popd	Cambia al directorio colocado en la parte superior de la pila.
%dirs	Devuelve una lista que contiene la pila de directorios actual.
%dhist	Imprime el historial de directorios visitados.
%env	Devuelve las variables de entorno del sistema como un diccionario.
%matplotlib	Configura las opciones de integración de matplotlib.

Comandos de shell y alias

Si se inicia una línea en IPython con un signo de exclamación ! (denominado *bang* en IPython), se le indica a IPython que ejecute todo lo que haya después de dicho signo en el shell del sistema. Esto significa que se pueden borrar archivos (utilizando rm o del, dependiendo del sistema operativo que se tenga), cambiar directorios o ejecutar cualquier otro proceso.

Es posible almacenar en una variable la salida de consola de un comando del shell asignando a la variable la expresión con un signo ! delante. Por ejemplo, en mi máquina Linux conectada a internet mediante Ethernet, puedo obtener mi dirección IP como una variable Python:

```
In [1]: ip_info = !ifconfig wlan0 | grep "inet "
In [2]: ip_info[0].strip()
Out[2]: 'inet addr:10.0.0.11 Bcast:10.0.0.255 Mask:255.255.255.0'
```

El objeto Python devuelto ip_info es realmente del tipo lista personalizada y contiene varias versiones de la salida de consola.

IPython puede también sustituir en Python valores definidos en el entorno actual cuando se utiliza !. Para ello, colocamos delante del nombre de variable el signo del dólar \$:

```
In [3]: foo = 'test'*  
In [4]: !ls $foo  
test4.py test.py test.xml
```

La función mágica %alias puede definir atajos personalizados para comandos del shell. Como ejemplo:

```
In [1]: %alias ll ls -l  
In [2]: ll /usr  
total 332  
  
drwxr-xr-x 2 root root 69632 2012-01-29 20:36 bin/  
drwxr-xr-x 2 root root 4096 2010-08-23 12:05 games/  
drwxr-xr-x 123 root root 20480 2011-12-26 18:08 include/  
drwxr-xr-x 265 root root 126976 2012-01-29 20:36 lib/  
drwxr-xr-x 44 root root 69632 2011-12-26 18:08 lib32/  
lrwxrwxrwx 1 root root 3 2010-08-23 16:02 lib64 -> lib/  
drwxr-xr-x 15 root root 4096 2011-10-13 19:03 local/  
drwxr-xr-x 2 root root 12288 2012-01-12 09:32 sbin/  
drwxr-xr-x 387 root root 12288 2011-11-04 22:53 share/  
drwxrwsr-x 24 root src 4096 2011-07-17 18:38 src/
```

Se pueden ejecutar varios comandos, igual que en la línea de comandos, separándolos por signos de punto y coma:

```
In [558]: %alias test_alias (cd examples; ls; cd ..)  
In [559]: test_alias  
macrodata.csv spx.csv tips.csv
```

Se puede observar que IPython «olvida» cualquier alias definido interactivamente tan pronto como la sesión se cierra. Para crear alias permanentes, será necesario utilizar el sistema de configuración.

Sistema de marcado a directorios

IPython tiene un sistema de marcado a directorios que permite guardar alias de directorios de uso habitual, de forma que se pueda saltar a ellos fácilmente. Por ejemplo, supongamos que queremos crear un marcador que apunte al material complementario de este libro:

```
In [6]: %bookmark py4da /home/wesm/code/pydata-book
```

Una vez hecho esto, al utilizar la función mágica %cd, podemos usar cualquier marcador que hayamos definido:

```
In [7]: cd py4da  
(bookmark:py4da) -> /home/wesm/code/pydata-book  
/home/wesm/code/pydata-book
```

Si el nombre de un marcador entra en conflicto con el nombre incluido en el directorio de trabajo actual, podemos emplear la bandera -b para omitirlo y utilizar la ubicación del marcador. Si se usa la opción -l con %bookmark, se listan todos los marcadores:

```
In [8]: %bookmark -l  
Current bookmarks:  
py4da -> /home/wesm/code/pydata-book-source
```

Los marcadores, a diferencia de los alias, se conservan de forma automática entre sesiones IPython.

B.5 Herramientas de desarrollo de software

Además de ser un entorno cómodo para computación interactiva y exploración de datos, IPython puede también ser un compañero útil para el desarrollo de software en general. En aplicaciones de análisis de datos, es importante tener en primer lugar un código correcto. Por suerte, IPython ha puesto mucho cuidado en integrar y mejorar el depurador pdb interno de Python. En segundo lugar, queremos que nuestro código sea rápido. Para ello, IPython ofrece prácticas herramientas integradas de medición y perfilado de código. Voy a ofrecer aquí un resumen de dichas herramientas.

Depurador interactivo

El depurador de IPython mejora pdb con completado por tabulación, resultado de sintaxis y contexto para cada línea en los rastreos de excepciones. Uno de los mejores momentos para depurar código es justo después de producirse un error. El comando %debug, cuando se introduce inmediatamente después de una excepción, invoca el depurador «post mortem» y nos deja en el marco de pila cuando se produce la excepción:

```
In [2]: 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:

In [3]: %debug
> /home/wesm/code/pydata-book/examples/ipython_bug.py(9)throws_an_exception()

    8             b = 6
-->  9             assert(a + b == 10)
   10
ipdb>

```

Una vez dentro del depurador, es posible ejecutar código Python arbitrario y explorar todos los objetos y datos («mantenidos con vida» por el intérprete) dentro de cada marco de pila. De forma predeterminada se empieza en el nivel inferior, donde se produjo el error. Escribiendo u (up: arriba) y d (down: abajo), se puede cambiar entre los niveles del stack trace (o informe de pila):

```

ipdb> u
> /home/wesm/code/pydata-book/examples/ipython_bug.py(13)calling_things()

    12         works_fine()
-->  13         throws_an_exception()
   14

```

Ejecutar el comando %pdb hace que IPython invoque automáticamente al depurador después de cualquier excepción, un modo que muchos usuarios encontrarán útil.

También resulta práctico utilizar el depurador al desarrollar código, especialmente cuando se necesita establecer un punto de interrupción (o *step*) o ir avanzando poco a poco a lo largo de la ejecución de una función o script para examinar su comportamiento en cada paso. Hay varias formas de hacer esto. La primera es utilizando %run con la bandera -d, que invoca al depurador antes de ejecutar cualquier fragmento del código que se haya pasado. Se debe escribir inmediatamente s (de *step*) para introducir el script:

```

In [5]: run -d examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()
ipdb> s
-Call-
> /home/wesm/code/pydata-book/examples/ipython_bug.py(1)<module>()

1-->  1         def works_fine():
      2             a = 5
      3             b = 6

```

Después de esto, cada uno decide cómo desea actuar al avanzar por el archivo. Por ejemplo, en la excepción anterior podríamos establecer un punto de interrupción justo antes de llamar a la función `works_fine`, y ejecutar el código hasta alcanzar dicha interrupción escribiendo `c` (de *continue* continuar):

```
ipdb> b 12
ipdb> c

> /home/wesm/code/pydata-book/examples/ipython_bug.py(12)calling_things()

      11
2->  12 works_fine()
      13           throws_an_exception()
```

En este momento se puede pasar a `works_fine()` o ejecutarlo escribiendo `n` (de *next*: siguiente) para avanzar a la siguiente línea:

```
ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(13)calling_things()

      2           12 works_fine()
-->  13           throws_an_exception()
      14
```

A continuación podemos parar en `throws_an_exception`, avanzar a la línea en la que se produce el error y mirar las variables del ámbito. Tengamos en cuenta que los comandos del depurador tienen prioridad sobre los nombres de las variables; en estos casos, ponemos un signo ! delante de las variables para examinar su contenido:

```
ipdb> s
-Call-
> /home/wesm/code/pydata-book/examples/ipython_bug.py(6)throws_an_exception()

      5
-->  6           def throws_an_exception():
      7             a = 5

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(7)throws_an_exception()

      6           def           throws_an_exception():
-->  7             a = 5
      8             b = 6

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(8)throws_an_exception()

      7             a = 5
-->  8             b = 6
      9             assert(a + b == 10)
```

```

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(9) throws_an_exception()

     8             b = 6
→   9             assert(a + b == 10)
    10

ipdb> !a
5
ipdb> !b
6

```

Según mi experiencia, desarrollar un buen dominio del depurador interactivo requiere tiempo y práctica. La tabla B.4 lista un catálogo completo de los comandos del depurador. Si se utiliza habitualmente un IDE, quizás al principio el depurador dirigido al terminal pueda parecer un poco despiadado, pero con el tiempo eso cambiará. Algunos IDE de Python tienen excelentes depuradores GUI, de modo que prácticamente todos los usuarios encontrarán algo que les funcione.

Tabla B.4. Comandos del depurador de Python.

Comando	Acción
h(elp)	Muestra la lista de comandos.
help <i>comando</i>	Muestra la documentación de <i>comando</i> .
c(ontinue)	Reanuda la ejecución del programa.
q(uit)	Sale del depurador sin ejecutar más código.
b(reak) <i>número</i>	Establece un punto de interrupción en <i>número</i> en el archivo actual.
b <i>ruta/archivo.py:número</i>	Establece un punto de interrupción en la línea <i>número</i> del archivo especificado.
s(tep)	Pasa a la llamada a la función.
n(ext)	Ejecuta la línea actual y avanza a la siguiente en el nivel actual.
u(p)/d(own)	Sube y baja dentro de la pila de llamadas a función.
a(rgs)	Muestra los argumentos de la función actual.
debug <i>sentencia</i>	Invoca <i>sentencia</i> en el nuevo depurador (recursivo).
l(ist) <i>sentencia</i>	Muestra la posición actual y el contexto en el nivel de pila actual.
w(here)	Imprime el informe de pila completo con contexto en la posición actual.

Otras formas de utilizar el depurador

Hay otras dos maneras útiles de invocar al depurador. La primera es utilizando una función `set_trace` especial (llamada así por `pdb.set_trace`), que es básicamente el método económico. Aquí hay un par de fórmulas que quizás convenga guardar para su uso habitual (añadiéndolas quizás también al perfil de IPython, como yo hago):

```

from IPython.core.debugger import Pdb
def set_trace():

```

```

Pdb(.set_trace(sys._getframe().f_back)

def debug(f, *args, **kwargs):

    pdb = Pdb()
    return pdb.runcall(f, *args, **kwargs)

```

La primera función, `set_trace`, ofrece una forma cómoda de poner un punto de interrupción en algún lugar del código. Se puede usar un `set_trace` en cualquier parte del código que se desee detener temporalmente para examinarlo con más atención (por ejemplo, justo antes de que se produzca una excepción):

```

In [7]: run examples/ipython_bug.py
> /home/wesm/code/pydata-book/examples/ipython_bug.py(16)calling_things()

```

```

      15      set_trace()
-->  16      throws_an_exception()
      17

```

Escribir `c` (*continue*) hará que el código se siga ejecutando normalmente sin ningún problema.

La función `debug` que acabamos de ver permite invocar al depurador interactivo fácilmente en una llamada a función arbitraria. Supongamos que hemos escrito una función como la siguiente, y queremos recorrer su lógica:

```

def f(x, y, z=1):

    tmp = x + y
    return tmp / z

```

Normalmente, usando `f` quedaría algo así: `f(1, 2, z=3)`. Sin embargo, para recorrer `f`, le pasamos `f` a `debug` como primer argumento, seguido de los argumentos posicional y de palabra clave que se le han de pasar a `f`:

```

In [6]: debug(f, 1, 2, z=3)
> <ipython-input>(2)f()

```

```

      1      def f(x, y, z):
-->  2          tmp = x + y
      3          return tmp / z

ipdb>

```

Estas dos fórmulas me han ahorrado mucho tiempo a lo largo de los años.

Por último, el depurador se puede usar junto con `%run`. Ejecutar un código con `%run -d` nos lleva directamente al depurador, preparados para establecer cualquier punto de interrupción e iniciar la ejecución:

```

In [1]: %run -d examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:1

```

```
NOTE: Enter 'c' at the ipdb> prompt to start your script.  
> <string>(1)<module>()  
ipdb>
```

Añadir -b con un número de línea inicia el depurador con un punto de interrupción ya establecido:

```
In [2]: %run -d -b2 examples/ipython_bug.py  
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:2  
NOTE: Enter 'c' at the ipdb> prompt to start your script.  
> <string>(1)<module>()  
ipdb> c  
> /home/wesm/code/pydata-book/examples/ipython_bug.py(2)works_fine()
```

```
1           def works_fine():  
1--> 2           a = 5  
      3           b = 6
```

```
ipdb>
```

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

Para aplicaciones de análisis de datos a gran escala o de larga ejecución, quizá sea interesante medir el tiempo de ejecución de los distintos componentes, de las sentencias individuales o de las llamadas a función. Nos puede venir bien disponer de un informe de las funciones que están necesitando más tiempo en un proceso complejo. Afortunadamente, IPython permite conseguir esta información de un modo práctico, al mismo tiempo que se desarrolla y prueba el código.

Medir el tiempo de ejecución del código a mano utilizando el módulo `time` interno y sus funciones, `time.clock` y `time.time`, suele ser tedioso y repetitivo, ya que hay que escribir el mismo código trillado y aburrido:

```
import time  
start = time.time()  
  
for i in range(iterations):  
    # aquí va algo de código  
  
elapsed_per = (time.time()-start) / iterations
```

Como esta es una operación tremadamente habitual, IPython tiene dos funciones mágicas, `%time` y `%timeit`, para automatizar este proceso.

La primera, `%time`, ejecuta una sentencia una vez, informando del tiempo de ejecución total. Supongamos que tenemos una larga lista de cadenas de texto y queremos comparar los distintos métodos de seleccionar todas las que empiecen por un determinado prefijo. Aquí muestro una lista de 600000 cadenas de texto y dos métodos idénticos para elegir solamente las que empiezan por 'foo':

```
# una lista de cadenas de texto muy larga  
In [11]: strings = ['foo', 'foobar', 'baz', 'qux',  
.....: 'python', 'Guido Van Rossum'] * 100000  
In [12]: method1 = [x for x in strings if x.startswith('foo')]
```

```
In [13]: method2 = [x for x in strings if x[:3] == 'foo']
```

Parece que deberían tener el mismo rendimiento, ¿verdad? Podemos comprobarlo para estar seguros utilizando %time:

```
In [14]: %time method1 = [x for x in strings if x.startswith('foo')]  
CPU times: user 52.5 ms, sys: 0 ns, total: 52.5 ms
```

Wall time: 52.1 ms

```
In [15]: %time method2 = [x for x in strings if x[:3] == 'foo']  
CPU times: user 65.3 ms, sys: 0 ns, total: 65.3 ms
```

Wall time: 64.8 ms

La expresión `Wall time`, abreviatura de «wall-clock time» (tiempo de reloj de pared) es el número que más nos interesa aquí. A partir de estos tiempos, podemos deducir que hay diferencia en el rendimiento, pero también que no es una medida muy precisa. Si probamos a aplicar %time a estas sentencias varias veces, descubriremos que los resultados son distintos. Para obtener una medición más exacta, es mejor emplear la función mágica `%timeit`. Dada una sentencia arbitraria, tiene una heurística para ejecutar una sentencia varias veces y producir así un tiempo de ejecución medio más preciso (estos resultados pueden diferir en el sistema de cada usuario):

```
In [563]: %timeit [x for x in strings if x.startswith('foo')]  
10 loops, best of 3: 159 ms per loop
```

```
In [564]: %timeit [x for x in strings if x[:3] == 'foo']  
10 loops, best of 3: 59.3 ms per loop
```

Este ejemplo, aparentemente inocuo, ilustra que merece la pena comprender las características de rendimiento de la librería estándar de Python, de NumPy, pandas y de las otras librerías empleadas en este libro. En aplicaciones de análisis de datos a gran escala, esos milisegundos empezarán a sumar.

La otra función mágica, `%timeit`, es especialmente útil para analizar sentencias y funciones con tiempos de ejecución muy cortos, incluso a nivel de microsegundos (millonésimas de segundo) o nanosegundos (billonésimas de segundo). Quizá puedan parecer cantidades de tiempo insignificantes, pero por supuesto una función de 20 microsegundos invocada un millón de veces tarda 15 segundos más que una función de 5 microsegundos. En el ejemplo anterior, podríamos comparar directamente las dos operaciones de cadena de texto para comprender sus características de rendimiento:

```
In [565]: x = 'foobar'
```

```
In [566]: y = 'foo'
```

```
In [567]: %timeit x.startswith(y)  
1000000 loops, best of 3: 267 ns per loop
```

```
In [568]: %timeit x[:3] == y  
10000000 loops, best of 3: 147 ns per loop
```

Perfilado básico: %prun y %run -p

El perfilado de código está estrechamente relacionado con la medición del tiempo de ejecución, salvo porque su objetivo es determinar en qué se emplea el tiempo. La principal herramienta de perfilado de Python es el módulo `cProfile`, que no es en absoluto específico de IPython. Este módulo ejecuta un programa u otro bloque de código arbitrario controlando simultáneamente el tiempo que se emplea en

cada función. Una forma habitual de usar cProfile es en la línea de comandos, ejecutando un programa entero y obteniendo el tiempo agregado por función. Supongamos que tenemos un código que realiza una cierta álgebra lineal en un bucle (calculando los valores únicos máximos y absolutos de una serie de 100 × 100 matrices):

```
import numpy as np
from numpy.linalg import eigvals

def run_experiment(niter=100):

    K = 100
    results = []
    for _ in range(niter):
        mat = np.random.standard_normal((K, K))
        max_eigenvalue = np.abs(eigvals(mat)).max()
        results.append(max_eigenvalue)

    return results
some_results = run_experiment()
print('Largest one we saw: {}'.format(np.max(some_results)))
```

Podemos ejecutar este código mediante cProfile tecleando lo siguiente en la línea de comandos:

```
python -m cProfile cprof_example.py
```

Al escribir esto, descubrimos que la salida se obtiene clasificada por nombre de función, lo que dificulta un poco hacerse una idea de en qué se emplea el tiempo, de modo que resulta útil especificar una orden de clasificación con la bandera -s:

```
$ python -m cProfile -s cumulative cprof_example.py
Largest one we saw: 11.923204422

15116 function calls (14927 primitive calls) in 0.720 seconds
Ordered by: cumulative time
```

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.001	0.001	0.721	0.721	cprof_example.py:1(<module>)
100	0.003	0.000	0.586	0.006	linalg.py:702(eigvals)
200	0.572	0.003	0.572	0.003	{numpy.linalg.lapack_lite.dgeev}
1	0.002	0.002	0.075	0.075	__init__.py:106(<module>)
100	0.059	0.001	0.059	0.001	{method 'randn'}
1	0.000	0.000	0.044	0.044	add_newdocs.py:9(<module>)
2	0.001	0.001	0.037	0.019	__init__.py:1(<module>)
2	0.003	0.002	0.030	0.015	__init__.py:2(<module>)
1	0.000	0.000	0.030	0.030	type_check.py:3(<module>)
1	0.001	0.001	0.021	0.021	__init__.py:15(<module>)
1	0.013	0.013	0.013	0.013	numeric.py:1(<module>)
1	0.000	0.000	0.009	0.009	__init__.py:6(<module>)
1	0.001	0.001	0.008	0.008	__init__.py:45(<module>)
262	0.005	0.000	0.007	0.000	function_base.py:3178(add_newdoc)
100	0.003	0.000	0.005	0.000	linalg.py:162(_assertFinite)

```
...
```

Solamente se muestran las primeras 15 filas del resultado. Es más fácil de leer explorando hacia abajo la columna cumtime para ver cuánto tiempo total se empleó dentro de cada función. Tengamos en cuenta que si una función llama a otra función cualquiera, el reloj no deja de correr. El módulo cProfile registra el tiempo inicial y final de cada llamada a función y utiliza estos datos para producir la medición del tiempo de ejecución.

Además del uso de la línea de comandos, cProfile puede utilizarse también en programación para perfilar bloques de código arbitrario sin tener que ejecutar un nuevo proceso. IPython dispone de una cómoda interfaz para esto utilizando el comando %prun y la opción -p de %run. El comando %prun admite las mismas «opciones de línea de comando» que cProfile pero perfilará una sentencia Python arbitraria en lugar de un archivo .py completo:

```
In [4]: %prun -l 7 -s cumulative run_experiment()

4203 function calls in 0.643 seconds

Ordered by: cumulative time
List reduced from 32 to 7 due to restriction <7>

ncalls  tottime  percall  cumtime  percall   filename:lineno(function)
      1    0.000    0.000    0.643    0.643   <string>:1(<module>)
      1    0.001    0.001    0.643    0.643   cprof_example.py:4(run_experiment)
    100    0.003    0.000    0.583    0.006   linalg.py:702(eigvals)
    200    0.569    0.003    0.569    0.003   {numpy.linalg.lapack_lite.dgeev}
    100    0.058    0.001    0.058    0.001   {method 'randn'}
    100    0.003    0.000    0.005    0.000   linalg.py:162(_assertFinite)
    200    0.002    0.000    0.002    0.000   {method 'all' of 'numpy.ndarray'}
```

De forma similar, llamar a %run -p -s cumulative cprof_example.py tiene el mismo efecto que el método de línea de comando, salvo que nunca hay que salir de IPython.

En el notebook de Jupyter se puede usar la función mágica %%prun (dos signos de porcentaje) para perfilar un bloque de código entero. Así, se abre una ventana distinta con el resultado del perfilado. Esto puede resultar útil para obtener quizás respuestas rápidas a preguntas como «¿por qué ese código tardó tanto en ejecutarse?».

Hay otras herramientas disponibles que facilitan la comprensión de la realización de perfilados cuando se utiliza IPython o Jupyter. Una de ellas es SnakeViz (<https://github.com/jiffyclub/snakeviz>), que produce una visualización interactiva de los resultados del perfilado utilizando D3.js.

Perfilar una función línea a línea

En algunos casos, la información obtenida de %prun (u otro método de perfilado basado en cProfile) puede no contarlo todo sobre el tiempo de ejecución de una función, o puede ser tan compleja que los resultados, agregados por nombre de función, resulten difíciles de interpretar. Para un caso como este disponemos de una pequeña librería llamada line_profiler (que puede conseguirse mediante PyPI o a través de una de las herramientas de gestión de paquetes). Contiene una extensión de IPython que habilita una nueva función mágica %lprun, encargada de realizar un perfilado línea a línea de una o

varias funciones. Esta extensión se puede habilitar modificando la configuración de IPython (véase la documentación de IPython o la sección dedicada a la configuración más adelante en este apéndice) para que incluya la siguiente línea:

```
# Una lista de nombres de módulo punteados de extensiones IPython para cargar.  
c.InteractiveShellApp.extensions = ['line_profiler']
```

También se puede ejecutar el comando:

```
%load_ext line_profiler
```

El atributo `line_profiler` se puede utilizar en programación (véase la documentación completa), pero quizás es más potente cuando se utiliza interactivamente en IPython. Supongamos que tenemos un módulo `prof_mod` con el siguiente código que realiza algunas operaciones con arrays NumPy (para reproducir este ejemplo, basta con poner este código en un nuevo archivo `prof_mod.py`):

```
from numpy.random import randn  
def add_and_sum(x, y):  
  
    added = x + y  
    summed = added.sum(axis=1)  
    return summed  
  
def call_function():  
  
    x = randn(1000, 1000)  
    y = randn(1000, 1000)  
    return add_and_sum(x, y)
```

Si queremos comprender el rendimiento de la función `add_and_sum`, `%prun` nos da lo siguiente:

```
In [569]: %run prof_mod  
In [570]: x = randn(3000, 3000)  
In [571]: y = randn(3000, 3000)  
In [572]: %prun add_and_sum(x, y)  
4 function calls in 0.049 seconds
```

Ordered by: internal time					
ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.036	0.036	0.046	0.046	prof_mod.py:3(add_and_sum)
1	0.009	0.009	0.009	0.009	{method 'sum' of 'numpy.ndarray'}
1	0.003	0.003	0.049	0.049	<string>:1(<module>)

Esto no es especialmente esclarecedor. Con la extensión `line_profiler` de IPython activada, está disponible un nuevo comando `%lprun`. La única diferencia de uso es que debemos indicar a `%lprun` la función o funciones que deseamos perfilar. La sintaxis general es:

```
%lprun -f func1 -f func2 statement_to_profile
```

En este caso, queremos perfilar `add_and_sum`, así que ejecutamos:

```
In [573]: %lprun -f add_and_sum add_and_sum(x, y)
Timer unit: 1e-06 s
```

```
File: prof_mod.py
Function: add_and_sum at line 3
Total time: 0.045936 s
```

Line #	Hits	Time	Per Hit	% Time	Line Contents
3					def add_and_sum(x, y):
4	1	36510	36510.0	79.5	added = x + y
5	1	9425	9425.0	20.5	summed = added.sum(axis=1)
6	1	1	1.0	0.0	return summed

Esto puede resultar mucho más fácil de interpretar. En este caso, perfilamos la misma función empleada en la sentencia. Mirando el código de módulo anterior, podríamos llamar a `call_function` y perfilar esto además de `add_and_sum`, para obtener así una imagen completa del rendimiento del código:

```
In [574]: %lprun -f add_and_sum -f call_function call_function()
Timer unit: 1e-06 s
```

```
File: prof_mod.py
Function: add_and_sum at line 3
Total time: 0.005526 s
```

Line #	Hits	Time	Per Hit	% Time	Line Contents
3					def add_and_sum(x, y):
4	1	4375	4375.0	79.2	added = x + y
5	1	1149	1149.0	20.8	summed = added.sum(axis=1)
6	1	2	2.0	0.0	return summed

```
File: prof_mod.py
Function: call_function at line 8
Total time: 0.121016 s
```

Line #	Hits	Time	Per Hit	% Time	Line Contents
8					def call_function():
9	1	57169	57169.0	47.2	x = randn(1000, 1000)
10	1	58304	58304.0	48.2	y = randn(1000, 1000)
11	1	5543	5543.0	4.6	return add_and_sum(x, y)

Como regla general, tiendo a preferir `%prun` (`cProfile`) para realizar perfilado «macro», y `%lprun` (`line_profiler`) para perfilado «micro». Merece la pena entender bien ambas herramientas.



La razón de que se deban especificar de manera explícita los nombres de las funciones que se desea perfilar con `%lprun` es que la sobrecarga de «rastrear» el tiempo de ejecución de cada línea es importante. Las funciones de rastreo que no son de interés tienen el potencial de alterar significativamente los resultados del perfil.

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

Escribir código de una forma que lo haga cómodo de desarrollar, depurar, en definitiva, utilizar interactivamente, puede ser un cambio de paradigma para muchos usuarios. Hay detalles

procedimentales, como la recarga de código, que pueden requerir ciertos ajustes, además de cuestiones de estilo de codificación.

Por tanto, implementar la mayoría de las estrategias descritas en esta sección es más un arte que una ciencia, y requerirá una cierta experimentación por parte del usuario para determinar una forma de escribir código Python que sea eficaz. En última instancia lo que queremos es estructurar el código de modo que sea práctico de utilizar iterativamente y que permita explorar los resultados de la ejecución de un programa o función con el menor esfuerzo posible. He encontrado software diseñado teniendo IPython en mente con el fin de que resulte más sencillo de manejar que el código destinado únicamente a ser ejecutado como una aplicación de línea de comando independiente. Esto es especialmente importante cuando algo va mal y hay que diagnosticar un error en el código que uno mismo u otra persona podría haber escrito meses o años antes.

Recargar dependencias de módulo

En Python, al escribir `import some_lib` se ejecuta el código de `some_lib`, y todas las variables, funciones e importaciones definidas en su interior se almacenan en el espacio de nombres del módulo `some_lib` recién creado. La siguiente vez que se utilice `import some_lib`, se obtiene una referencia al espacio de nombres de módulo existente. La posible dificultad del desarrollo de código interactivo IPython ocurre cuando, por ejemplo, se ejecuta con `%run` un script que depende de otro módulo en el que quizás se han hecho cambios. Supongamos que tenemos el siguiente código en `test_script.py`:

```
import some_lib
x = 5
y = [1, 2, 3, 4]
result = some_lib.get_answer(x, y)
```

Si ejecutáramos `%run test_script.py` y después modificáramos `some_lib.py`, la próxima vez que ejecutáramos `%run test_script.py` seguiríamos obteniendo la versión anterior de `some_lib.py` debido al sistema modular «de carga única» de Python. Este comportamiento difiere del de otros entornos de análisis de datos, como MATLAB, que propaga automáticamente los cambios en el código⁵. Para solucionar esto hay dos opciones. La primera es usar la función `reload` del módulo `importlib` de la librería estándar:

```
import some_lib
import importlib

importlib.reload(some_lib)
```

Con esto se intenta proporcionar una copia limpia de `some_lib.py` cada vez que se ejecuta `test_script.py` (pero en algunas situaciones esto no pasa). Obviamente, si las dependencias son más profundas, podría ser complicado estar insertando usos de recarga por todas partes. Para solucionar este problema, IPython tiene una función especial `reload` (no mágica) para la recarga «profunda» (recursiva) de módulos. Si ejecutáramos `some_lib.py` y después usaríamos `dre.load(some_lib)`, intentaría recargar `some_lib` además de todas sus dependencias. Esto no funcionará en todos los casos, lamentablemente, pero cuando lo hace, es mejor reiniciar IPython.

Consejos de diseño de código

No hay una receta sencilla para esto, pero sí puedo ofrecer algunos principios que he encontrado efectivos en mi trabajo.

Mantener activos los objetos y datos relevantes

No es poco habitual ver un programa escrito para la línea de comandos con una estructura similar a esta:

```
from my_functions import g

def f(x, y):
    return g(x + y)

def main():

    x = 6
    y = 7.5
    result = x + y

    if __name__ == '__main__':
        main()
```

Poniendo atención se puede observar lo que podría ir mal si ejecutáramos este código en IPython, ¿verdad? Una vez hecho, ninguno de los resultados u objetos definidos en la función `main` será accesible en el shell de IPython. Una forma mejor de hacer esto es ejecutar el código que haya en `main` directamente en el espacio de nombres global del módulo (o en el bloque `if __name__ == '__main__':`, si queremos que el módulo pueda también importarse). De este modo, cuando se aplica `%run` para ejecutar el código, se podrán ver todas las variables definidas en `main`. Esto equivale a definir variables de máximo nivel en celdas del notebook de Jupyter.

Plano es mejor que anidado

El código profundamente anidado me hace pensar en las capas de una cebolla. Al probar o depurar una función, ¿cuántas capas de la cebolla se deben pelar para alcanzar el código deseado? La idea de que «plano es mejor que anidado» es una parte del pensamiento zen de Python, y se aplica generalmente también al desarrollo de código para su uso interactivo. Crear funciones y clases tan disociadas y modulares como sea posible las hace más fáciles de probar (si estamos escribiendo código de prueba), depurar y utilizar interactivamente.

Superar el miedo a los archivos más largos

Si alguno de mis lectores viene de Java (u otro lenguaje similar), quizás le hayan dicho que mantenga cortos los archivos. En muchos lenguajes es un buen consejo; una gran longitud suele ser sinónimo de «mal código», y un indicador de que puede ser necesario realizar refactorización o reorganización. No obstante, mientras se desarrolla código con IPython, trabajar con diez archivos pequeños (digamos de 100 líneas cada uno) pero interconectados causará probablemente más problemas en general que tener dos o tres archivos más largos. Menos archivos significa menos módulos que recargar, y también menos

saltos entre archivos durante la edición. Yo he descubierto que mantener módulos más grandes, cada uno con una elevada cohesión interna (el código se relaciona con resolver los mismos tipos de problemas), resulta mucho más útil y «pitónico». Tras iterar hacia una solución, por supuesto que en ocasiones tendrá sentido refactorizar archivos más grandes en otros más pequeños.

Obviamente, no estoy a favor de llevar este argumento al extremo, que sería poner todo el código en un solo archivo monstruoso. Encontrar un módulo y una estructura de paquetes razonables e intuitivos para una base de código grande suele requerir un poco de esfuerzo, pero es especialmente importante acertar en los equipos. Cada módulo debería tener cohesión interna, y debería ser tan obvio como sea posible el lugar en el que encontrar funciones y clases responsables de cada área de funcionalidad.

B.7 Funciones avanzadas de IPython

Utilizar a fondo el sistema IPython puede lograr que escribamos código de un modo ligeramente distinto, o bien que profundicemos en la configuración.

Perfiles y configuración

La mayor parte de las facetas del aspecto (colores, prompt, espacio entre líneas, etc.) y el comportamiento de los entornos IPython y Jupyter pueden variarse a voluntad mediante un extenso sistema de configuración. Estas son algunas de las cosas que se puede hacer con este sistema:

- Cambiar el esquema de color.
- Cambiar el aspecto de los prompts de entrada y salida, o eliminar la línea blanca después de `out` y antes del siguiente prompt `In`.
- Ejecutar una lista arbitraria de sentencias Python (por ejemplo, importaciones que se utilizan todo el tiempo o cualquier otra cosa que queramos que ocurra cada vez que se lanza IPython).
- Habilitar las extensiones de IPython siempre conectadas, como la función mágica `%lprun` de `line_profiler`.
- Habilitar las extensiones de Jupyter.
- Definir funciones mágicas o alias de sistema propios.

Las configuraciones del shell de IPython se especifican en archivos `ipython_config.py` especiales, que se suelen encontrar en el directorio `.ipython/` del directorio raíz. La configuración se lleva a cabo basándose en un determinado perfil. Al iniciar IPython de la forma habitual se carga, por defecto, el perfil predeterminado, almacenado en el directorio `profile_default`. Así, en mi SO Linux, la ruta completa a mi configuración predeterminada de IPython es la siguiente:

```
/home/wesm/.ipython/profile_default/ipython_config.py
```

Para inicializar este archivo en otro sistema, hay que ejecutar esto en el terminal:

```
ipython profile create default
```

Les ahorraré los detalles del contenido de este archivo. Por suerte, incluye comentarios que describen para qué sirve cada opción de configuración, de modo que dejo a mis lectores que lo trasteen y personalicen. Otra función adicional útil es que es posible tener varios perfiles. Supongamos que queremos tener una configuración IPython alternativa adaptada a una determinada aplicación o proyecto. Crear un nuevo perfil implica escribir lo siguiente:

```
ipython profile create secret_project
```

Una vez hecho esto, editamos los archivos de configuración en el recién creado directorio `profile_secret_project` y lanzamos a continuación IPython, de este modo:

```
$ ipython --profile=secret_project
```

```
Python 3.8.0 | packaged by conda-forge | (default, Nov 22 2019, 19:11:19)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.22.0 - An enhanced Interactive Python. Type '?' for help.
```

```
IPython profile: secret_project
```

Como siempre, la documentación en línea de IPython es un recurso excelente para obtener más información sobre perfiles y configuración.

La configuración para Jupyter es ligeramente distinta porque se pueden utilizar sus notebooks con lenguajes distintos a Python. Para crear un archivo de configuración análogo para Jupyter, ejecutamos:

```
jupyter notebook --generate-config
```

De este modo se escribe un archivo de configuración predeterminado en el directorio `.jupyter/jupyter_notebook_config.py` del directorio raíz. Tras editarlo para adaptarlo a las necesidades particulares, se puede renombrar como un archivo diferente, como por ejemplo:

```
$ mv ~/.jupyter/jupyter_notebook_config.py ~/.jupyter/my_custom_config.py
```

Al lanzar Jupyter, se añade entonces el argumento `--config`:

```
jupyter notebook --config=~/.jupyter/my_custom_config.py
```

B.8 Conclusión

A medida que mis lectores trabajen en los ejemplos de código de este libro y mejoren sus habilidades como programadores de Python, les animo a seguir aprendiendo sobre los ecosistemas de IPython y Jupyter. Como estos proyectos se han diseñado para ayudar a la productividad del usuario, podrían descubrir herramientas que les permitan hacer su trabajo con mayor facilidad que utilizando el lenguaje Python y sus librerías computacionales por sí mismas.

En el sitio web de nbviewer (<https://nbviewer.org>) se pueden encontrar un montón de notebooks de Jupyter interesantes.

⁵ Como un módulo o paquete se puede importar en muchos sitios distintos de un determinado programa, la primera vez que se importa el código de un módulo, Python lo guarda en la caché, en lugar de ejecutarlo en el módulo en cada ocasión. De otro modo, la modularidad y la buena organización del código podrían provocar la posible falta de eficacia de una aplicación.

Título de la obra original: *Python for Data Analysis*

Traductor: Virginia Aranda González

Responsable editorial: Eugenio Tuya Feijoó

Revisión: Lidia Señarís

Adaptación de cubierta: Celia Antón Santos

Todos los nombres propios de programas, sistemas operativos, equipos hardware, etc. que aparecen en este libro son marcas registradas de sus respectivas compañías u organizaciones.

Authorized translation from English language edition published with the authorization of O'Reilly Media, Inc.

Copyright © 2022 Wesley McKinney. All rights reserved.

Edición en formato digital: 2023

© EDICIONES ANAYA MULTIMEDIA (GRUPO ANAYA, S. A.), 2023

Calle Valentín Beato, 21

28037 Madrid

ISBN ebook: 978-84-415-4724-7

Está prohibida la reproducción total o parcial de este libro electrónico, su transmisión, su descarga, su descompilación, su tratamiento informático, su almacenamiento o introducción en cualquier sistema de repositorio y recuperación, en cualquier forma o por cualquier medio, ya sea electrónico, mecánico, conocido o por inventar, sin el permiso expreso escrito de los titulares del Copyright.